AD-A104 445 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
DESIGN AND ANALYSIS OF A MULTI-BACKEND DATABASE SYSTEM FOR PERF--ETC(U)
AUG 81 M J MENON
UNCLASSIFIED OSU-CISRC-TR-81-8 N00014-75-C-0573
NL

1 OF 2
AD
A104445

TECHNICAL REPORT SERIES

LEVEL II

12

AD A104445

DTIC FILE COPY

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

81 8 26 021

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

12 (OSU-CISRC-TR-81-8)

DESIGN AND ANALYSIS OF A MULTI-BACKEND DATABASE SYSTEM FOR PERFORMANCE IMPROVEMENT, FUNCTIONALITY EXPANSION AND CAPACITY GROWTH (PART II)

by

David K. Hsiao
and
M. Jaishankar Menon

Work performed under

Contract N00014-75-C-0573

Office of Naval Research

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

August 1981

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OSU-CISRC-TR-81-8	2. GOVT ACCESSION NO. AD-A104 445	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design and Analysis of A Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part II).		5. TYPE OF REPORT & PERIOD COVERED Technical Report.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David K. Hsiao M. Jaishankar Menon		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, Virginia 22217		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS [illegible]		12. REPORT DATE
		13. NUMBER OF PAGES 109
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 1021P	ONR, Pasadena

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Access Control Mechanisms, Statistical Access Control, Value-Dependent Access Control, Concurrency Control Mechanisms, Monolithic Consistency of a Partitioned Database, Permutable Requests, Compatible Requests, Incompletely-Specified Transactions, Simulation Models, A Measure of Performance, Performance Under Various Conditions

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is Part II of the design and analysis of a multiple back-end database management system, known as MDBS. This part consists of the remaining of four chapters of the design and analysis details. The first four chapters were issued a month ago as Part I. For an abstract of the entire work, the reader is to refer to the abstract of Part I.

An implementation team is pursuing the instrumentation of MDBS as designed and analyzed in Parts I and II. It is hoped that studies and testings of the experimental MDBS may be used to verify of our analytic and simulation findings on

the MDBS design and performance.

7L-182 Per on file

A

TABLE OF CONTENTS

TABLE OF CONTENTS continued

TABLE OF CONTENTS continued

Page

TABLE OF CONTENTS continued

LIST OF FIGURES

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Professor of Computer and Information Science, and conducted at the Laboratory for Database Systems Research of the Ohio State University. The Laboratory for Database Systems Research of the Ohio State University is funded by the Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and the Ohio State University (OSU) and consists of the staff, graduate students, undergraduate students and faculty for conducting research in database systems. The research contract was administered and monitored by the Ohio State University Research Foundation. This report is issued by the Computer and Information Science Research Center (CISRC).

5. THE PROCESS OF REQUEST EXECUTION WITH ACCESS CONTROL

From the previous chapter, we learn that directory management in MDBS consists of three major phases. In the first phase, MDBS determines the exact clusters which will satisfy the user request. In the second phase, MDBS accesses security information about the user to select for the user the authorized clusters among the clusters which have been determined in the first phase. In the third phase, MDBS determines the secondary memory addresses of the authorized clusters selected in the second phase. The first and third phases of directory management, known as, respectively, the descriptor search and address generation phases, have been described in detail in Chapter 4. In this chapter, we describe the second phase of directory management. The processing performed by MDBS during the second phase of directory management will be referred to as the access control phase of directory management.

A. The Authorization Step

In describing the descriptor search and address generation phases of directory management, we note that the database creator may specify a number of descriptors at the database-creation time. These descriptors were used by MDBS to form the clusters of the database. (See Chapter 3 for the definition and use of descriptors.) Similarly, for access control, the database creator may also specify additional information along with each descriptor at the database-creation time. The process of request execution with access control begins long before the request is issued by the user to MDBS. There is the initial step of authorization by the database creator. The authorization step requires the database creator to specify, for each potential user of the database, the specific data and intended operations which are authorized for the user. It is therefore possible that two users of the same database may be authorized by the database creator with different data and operations of the database. The role of MDBS is to receive, maintain and enforce the authorizations. All the information that may be specified by a database creator at the database-creation time is explained in Section 5.1. How the information specified by the database creator is used by MDBS to determine the clusters which are authorized for a given user among all the clusters requested by the user is described in Section 5.2. Only after such determination is completed can a request from a user be carried through in MDBS.

B. Three Types of Access Control - by Granules, by Statistics and by Values

Upon receiving a user request, MDBS begins the process of request execution with access control. In controlling access to the database, MDBS provides an access control mechanism with considerable capabilities. First, it protects clusters. In other words, only authorized clusters will be permitted for the user. Further, only authorized operations are performed on the clusters so authorized. Second, it protects attribute values in the records of a cluster. Thus, access control is now brought from the cluster and record level to the field (i.e., attribute-value pair) level. This provides a finer granularity of security. Third, it protects the statistics of the database by controlling the execution of requests which utilize the aggregate functions such as average, maximum, minimum and others. Thus, statistical security becomes an integral part of the access control mechanism. Finally, it protects data on the basis of the relationship of the attribute value of the data and the user of the data. For example, a user may be given access to the records of those and only those employees who are managed by the user. Obviously, this user is a manager. Note that such access control cannot be achieved by protecting the values of some attributes in the records using the fine granularity of security discussed earlier. This is because if we were to give all employees who are managers an access to those records, then we would make the records available to other managers. What is really needed is for this and only this manager to access records of his or her employees. In other words, the access control is dependent upon specific attribute values. The access control mechanism of MDBS provides value-dependent security. In Section 5.4, we will describe two extensions to the basic access control mechanism for statistical security and value-dependent security. In Section 5.5, we describe the process of execution of transactions (consisting of multiple requests) with access control. Finally, in Section 5.6, we consider how a database creator may modify the authorization that was specified at the database-creation time.

C. A New Mode of Operation - Precision Control by Multiple Back-ends

From our subsequent discussion of the access control mechanism of MDBS, we will note the following two characteristics. First, the access control mechanism operates in a distributed fashion across multiple back-ends rather than in the controller. This serves to alleviate the controller limitation problem. Second, we will note that, in MDBS, all access control checks are made prior to the retrieval of records from the secondary store. Consequently, access imprecision due to redundant record retrieval never occurs. Thus, the need to discard retrieved records due to security violations does not arise, since only authorized records are retrieved from the secondary memory. Let us assume, for example, that a user wishes to retrieve 21 records but that 5 of these 21 records are not authorized for the user. Then, a conventional access control mechanism will first retrieve the 21 records. It will examine these 21 records to eliminate the 5 unauthorized ones before providing the user with the 16 authorized ones. However, in MDBS, the new access control mechanism will ensure that only the 16 authorized records will be retrieved from the secondary memory. Consequently, the access precision, defined as the ratio of the number of authorized records vs. the number of retrieved records, is always absolute, i.e., equal to 1, for MDBS.

5.1 Access Control as Exercised by the Database Creator

In this section, we will describe the kinds of access control information which may be specified by the database creator *at the database-creation time*. We will also describe how the information is stored in MDBS. In order to control access, a database creator needs to specify who can perform what operations on which data in his or her database. What we, as system designers, would like to do is to provide the database creator with an effective, yet efficient, way to convey these three pieces of information to MDBS. It is straightforward for a database creator to specify two of the three pieces of information, i.e., to identify the users who are allowed to access the database and to specify for these users the kinds of accesses allowed (or disallowed) to the database. For specifying 'who', the database creator utilizes the user id provided by the MDBS. For specifying 'what', the database creator selects the intended access operations from the set of MDBS operations such as retrieve, insert and so on. The third piece of information, to specify 'which' portions of the database is to be authorized for the users may be more involved. Let us describe the method used in our system for specifying portions of the database to be controlled for access.

We begin by arguing that a cluster is an ideal unit for access control. In Chapter 4, we showed that a cluster is the basic unit of access in MDBS. This is because the clusters are formed in such a way that whenever a user requests some records of a cluster, there is a high likelihood that all the records in the cluster are needed together. Since all the records in a cluster are likely being accessed together, it makes sense to accord the same control to all the records belonging to the cluster. Hence, we decide that a cluster should be a unit of access control in MDBS.

However, a database creator is unaware of the formation of clusters by MDBS. The database creator is only aware of the descriptors which in turn induce the clusters. Therefore, the database creator is provided with additional means to specify access control information with respect to these descriptors. Since these descriptors are used to form the clusters, the access control information specified by the database creator with respect to the descriptors may be transformed by MDBS into access control information about the clusters of the database. Thus, we now have a method for controlling access to clusters. If a cluster consists of a single record, then MDBS may control access at the level of individual records. If a cluster consists of many records, then MDBS may control access to record aggregates. Finally, MDBS may control certain attribute values of the records in a cluster. It will be seen that our access control mechanism is capable of three levels of control. In the following paragraphs, we will describe the means with which the database creator may specify various levels of access control.

First of all, a creator of a database must decide the users who will be allowed to access the database, i.e., the users who are allowed to issue requests (from either a terminal or an application program) to the database using the data manipulation language (DML) provided in Chapter 3. Note that the database creator is also one of the users of the database. After deciding the number of users, p, allowed to access the database, the database creator will inform MDBS of this number. MDBS will then assign p user ids for these p users. The database creator will reserve one of these ids for his or her own use and pass on the remaining ids to the other (p-1) users of the database. A user trying to use MDBS from a terminal must first identify the user by supplying the assigned id. Similarly, a user submitting a program to be executed in MDBS must include the user id as a part of the program. In this way, any request received by MDBS, either from a user at a terminal or as a part of a user program, is associated with the user. We shall refer to this id as the _user id of the request_.

Having received the id of the p users of the database, the database creator now specifies a number of descriptors as described in Section 4.1.1. In addition, the creator specifies, together with each descriptor, p sets of field-level access controls, one for each user of the database. A field-level access control is either null or 'all' or a pair of the form

([attribute combination], disallowed access operation).

An attribute combination is one or more attributes of the database. A disallowed access operation is one of the set {No-Retrieve, No-Delete, No-Insert, No-Update}. In the sequel, we shall often refer to the attribute combination part of a field-level access control and to the disallowed access (operation) part of a field-level access control. Whenever the disallowed access part of a field-level access control is No-Delete or No-Insert, the attribute combination part is left unspecified. This is because we always insert and delete entire records in MDBS. An example of a descriptor and a set of field-level access controls for a user is as follows:

$(0 < \text{Salary} < 100)$ (([Job],No-Update), ([Name,Salary],No-Retrieve))

The meaning of this example is as follows. The user is not permitted to update the job information from salary records whose salaries are ranged exclusively between zero and 100. Nor is the user permitted to retrieve the name and salary information from the same salary records. More formally, MDBS prevents the user from any update of the value of the Job attribute in those records whose keywords (i.e., Salary attribute and Salary value pairs) are derivable from the descriptor $(0 < \text{Salary} < 100)$. Similarly, MDBS prevents the user to read values of the Name and Salary attributes from records whose keywords are derivable from the descriptor $(0 < \text{Salary} < 100)$. We also stated that a field-level access control could be either null or all which indicates, respectively, that either no access or all accesses are disallowed. This completes our description of the kinds of information that must be specified by the database creator at the database-creation time.

By way of an example, let us illustrate the kinds of information specified by a database creator. Consider the database of employee records shown in Figure 33 and let us assume that only two users, identified as user 1 and user 2, are allowed to access this database. We assume that user 1 is the database creator and that user 2 is the only other user allowed to access the database.

After having received the user ids (i.e., user 1 and user 2) from MDBS, the database-creator specifies a number of descriptors. Furthermore, correspond-

One database with six records and four attributes per record

R_1: (<Employee,1>, <Department,1>, <Salary,1000>, <Manager,1>)

R_2: (<Employee,2>, <Department,1>, <Salary,8000>, <Manager,1>)

R_3: (<Employee,3>, <Department,2>, <Salary,15000>, <Manager,3>)

R_4: (<Employee,4>, <Department,2>, <Salary,2000>, <Manager,3>)

R_5: (<Employee,5>, <Department,3>, <Salary,7000>, <Manager,5>)

R_6: (<Employee,6>, <Department,3>, <Salary,16000>, <Manager,5>)

Figure 33. A Sample Database for Illustrating Field-Level Access Control

ing to each descriptor, the database creator specifies two sets of field-level access controls for the two users of the database, respectively. This information specified by the database creator is stored in an augmented descriptor-to-descriptor-id table (DDIT). (See a description of DDIT in Section 4.1.1.) The augmented DDIT consists of p additional columns, where p is the number of users anticipated to access the database. Thus, each entry of DDIT now consists of (p + 2) fields. The first field contains a descriptor id and the second field contains a descriptor. The third field contains the set of field-level access controls for the first user corresponding to the descriptor that is in the second field. Similarly, the (p + 2)-th field contains the set of field-level access controls for the p-th user corresponding to the same descriptor. An augmented DDIT for the sample database of Figure 33 is shown in Figure 34. We see that four descriptors (identified as D1, D2, D3 and D4) have been specified for the database. We also see that two sets of field-level access controls have been specified for each of these descriptors. In this figure, the set consisting of only the null field-level access control is denoted by '-' which indicates that no access is disallowed. It is therefore clear that user 1, the database creator, is not disallowed any access. That is, he is allowed to access any portion of the database. User 2, on the other hand, may access the database only in a controlled manner. In referring to Figure 34 again, for example, he is not allowed to learn the names of the managers from records in Department 3. This is controlled by the descriptor D4. The set of field-level access controls for user 2 corresponding to descriptor D2 is shown to be 'All' in Figure 34. This is used to indicate that all accesses are disallowed for user 2 to salary records whose salaries are greater than 10,000.

We have now completely described the kinds of information specified by a database creator, and we have also indicated how this database-creator-specified information is stored in an augmented DDIT.

5.2 Determination and Organization of the Exact Access Control from the Database-Creator-Specified Information

Once the descriptors and corresponding sets of field-level access controls have been specified and stored, the clusters of the database may be formed by using the algorithm for cluster formation described in Section 4.1.1. The clusters formed for the sample database of Figure 33 are reflected in the

Descriptor id	Descriptor	Set of Field-Level Access Controls For User 1	Set of Field-Level Access Controls For User 2
D1	$(1000 \leq \text{Salary} \leq 10000)$	—	([Employee,Salary],No-Retrieve)
D2	$(10000 < \text{Salary} < \infty)$	—	All
D3	$(1 \leq \text{Department} \leq 2)$	—	([Manager],No-Update), ([Employee,Salary],No-Retrieve)
D4	$(\text{Department} = 3)$	—	([Manager],No-Retrieve)

'-' indicates that no access is disallowed

'All' indicates that all accesses are disallowed

Figure 34. The Augmented Descriptor-to-Descriptor-Id Table (DDIT) for the Sample Database of Figure 33

cluster definition table (CDT) as depicted in Figure 35.

5.2.1 Controlling Access to Authorized Clusters

In order to control access, we also need, corresponding to each cluster, p sets of field-level access controls, one for each user of the database. The set of field-level access controls for a user corresponding to a cluster indicates the disallowed accesses for the user to that cluster. We now describe how the set of field-level access controls for a user corresponding to a cluster is determined.

Let us assume that we wish to find the set of field-level access controls for user 2 on cluster 2. We first learn, from Figure 35, that cluster 2 is defined by the descriptors D1 and D4. Then, the set of field-level access controls for user 2 on cluster 2 is obtained as the union of the sets of field-level access controls for user 2 corresponding to descriptors D1 and D4. The set of field-level access controls for user 2 on descriptors D1 and D4 are obtained from the augmented DDIT (see Figure 34). Thus, their union can be taken to determine the set of field-level access controls for user 2 on cluster 2. In general, the set of field-level access controls for user x on cluster y is obtained as the union of the sets of field-level access controls for user x corresponding to each of the defining descriptors of cluster y. By repeating the above procedure for every cluster in the database and every user authorized for the database, the sets of field-level access controls for each user over every cluster may be determined.

5.2.2 Organizing and Storing Cluster Control Tables

Having determined the set of field-level access controls for each user over every cluster, we are now concerned with methods for organizing and storing the access control information. There are two possible techniques for organizing and storing the access control information. The first technique utilizes a separate cluster definition table for each user of the database. The table for a user will only contain the access control information of those clusters to which the user is allowed some access. This table contains one more column than the CDT described in Chapter 4. This additional column contains the set of field-level access controls for this user corresponding to the various clusters. For the examples developed in Figures 34 and 35, the augmented CDT for user 2 is shown in Figure 36. Note that there

Cluster id	Set of Descriptors	Records in the Cluster Defined by the Descriptor Set
1	{D1,D3}	R1,R2,R4
2	{D1,D4}	R5
3	{D2,D3}	R3
4	{D2,D4}	R6

Figure 35. The Cluster Definition Table (CDT)

Cluster id	Set of Descriptors	Records in the Cluster Defined by the Descriptor Set	Set of Field-Level Access Controls
1	{D1,D3}	R1,R2,R4	(([Manager],No-Update), ([Employee,Salary],No-Retrieve))
2	{D1,D4}	R5	(([Manager],No-Retrieve), ([Employee,Salary],No-Retrieve))

Figure 36. The Augmented Cluster Definition Table for User 2

are only two clusters in the augmented CDT for user 2, even though there are four clusters in the database. To summarize, such a technique requires MDBS to maintain one CDT for the entire database and an augmented CDT for each user.

A second tecnhique for organizing and storing the set of field-level access controls for each user over every cluster is to collect all the access control information for all the users in one centralized cluster definition table. In this case, the sets of field-level access controls over every cluster is stored in MDBS by augmenting the CDT with p columns, one for each user of the database. In our example, the CDT of Figure 35 is augmented by two columns as shown in Figure 37. This centralized CDT will then be used by MDBS during request execution.

Either technique may be used in MDBS for organizing or storing the set of field-level access controls for each user over every cluster. The first technique will generally be superior in terms of the speed of request execution. This is because the number of clusters in the augmented CDT for a user will generally be less than the number of clusters in the centralized CDT for all the users. Consequently, the first technique entails a search through a smaller table. This is what contributes to the speed superiority of the first technique.

The first technique is also superior in terms of ease of maintenance. For instance, if one of the users of the database is removed from the system, we may simply delete the corresponding augmented CDT. In the second technique, we need to remove all the field-level access controls corresponding to that user from the centralized CDT. Irrespective to the physical implementation of the centralized CDT, the removal of all the field-level access controls corresponding to a user from the centralized CDT can be no simpler than the deletion of a table. Thus, we conclude that the first technique should be chosen in MDBS for organizing and storing the sets of field-level access controls for each user over every cluster. That is, physically, a separate augmented CDT is maintained for each user of the database, in addition to the one CDT for the entire database which is also maintained by MDBS. Logically, however, we may still assume that the sets of field-level access controls for each user over every cluster are consolidated and maintained as a single centralized CDT. Such an assumption simplifies some of the ensuing discussion.

In the next section, we will describe the process of request execution with access control which makes use of the augmented or centralized CDT formed at the database-creation time.

Cluster id	Set of Descriptors	Records in the Cluster Defined by the Descriptor Set	Set of Field-Level Access Controls For User 1	Set of Field-Level Access Controls for User 2
1	{D1,D3}	R1,R2,R4	—	(([Manager],No-Update), ([Employee,Salary],No-Retrieve))
2	{D1,D4}	R5	—	(([Manager],No-Retrieve), ([Employee,Salary],No-Retrieve))
3	{D2,D3}	R3	—	All
4	{D2,D4}	R6	—	All

'-' indicates that no access is disallowed

'All' indicates that all accesses are disallowed

Figure 37. The Augmented CDT for the Sample Database of Figure 33

5.3 Request Execution With Fine Granularity of Access Control

Up to this point, all the events we have described take place at the database-creation time long before any request is being executed. Let us now describe the process of request execution with access control for insert and non-insert requests in MDBS. We recall again (see Chapter 4) that the process of request execution consists of the three phases of directory management followed by actual processing of records retrieved from the secondary memory. Let us refer to the last stage of request execution in MDBS, i.e., the one follows the three phases of directory management, as record processing. In Section 4.2.1, we referred to the first and third phases of directory management as the descriptor search phase and the address generation phase, respectively. We also referred to the processing performed during the descriptor search phase as descriptor processing and the processing performed during the address generation phase as address generation. In summary, the process of request execution in MDBS without access control consists of descriptor processing, address generation and record processing. When access control is required, the process of request execution has added complexity. It still consists of descriptor processing and record processing as described in Section 4.3. However, the address generation phase is expanded.

In Chapter 4, we stated that address generation is executed at each back-end and consists of two steps. In the first step, the corresponding descriptors produced by the descriptor processing are used in order to find the corresponding cluster (for an insert request) or the corresponding set of clusters (for a non-insert request). We also stated that in the second step the secondary memory addresses of the corresponding cluster or corresponding set of clusters are generated. When access control is enforced, address generation is again executed at each back-end, but it consists of one additional step. Let us discuss this step with other steps together. In the first step, the corresponding descriptors produced by the descriptor processing are used in order to find the corresponding cluster or set of clusters. This step has not been changed. In the second step, the authorized clusters are selected on the basis of the information provided in the augmented CDT. This is the new step. Finally, in the third step, the secondary memory addresses of the corresponding authorized cluster or corresponding set of authorized clusters are generated. In this final step, the amount of processing may be reduced because the number of authorized clusters is usually smaller than the number of relevant clusters. However, the processing logic does not change. To summarize our discussion, the process of request execution with access control is different from the process of request execution without access control because the former consists of one additional step during address generation. We now elaborate the processing performed in this additional step of address generation as follows.

5.3.1 Executing Insert Requests

In the case of an insert request, MDBS performs descriptor processing and the first step of address generation in order to determine the cluster k and the back-end b into which the record in the request is to be inserted. (See again Section 4.3.4.) After such determination, the controller sends the record to back-end b for insertion. The second (i.e., new) step of address generation is now activated. The back-end b will search its augmented CDT in order to determine if the user that issued the insert request is authorized with such an insert to cluster k. This is accomplished by looking up the entry in the row for cluster k and in the column for the user id of the user that issued the insert request. If this entry contains a field-level access control with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, the request is rejected. Otherwise, the insert request is permitted. Consequently, the back-end proceeds with the next step of address generation and final step of record processing (both of which have been described in Section 4.3.).

5.3.2 Executing Non-Insert Requests

Non-insert requests consist of retrieve, delete and update. Before we describe the process of request execution for these requests, we will discuss some implications of their execution in the context of access control.

We recall that the database creator specifies descriptors and several sets of corresponding field-level access controls. These field-level access controls indicate the disallowed access operations. Thus, for instance, the database creator may specify the following descriptor and corresponding set of field-level access controls for user 1.

$(0 < \text{Salary} \leq 100)$ ([Salary],No-Retrieve)

This indicates that user 1 is not permitted to retrieve the salary field from records containing keywords derived from the descriptor ($0 < \text{Salary} \leq 100$). However, the database creator has not indicated whether user 1 is allowed to update, delete and insert such records. Therefore, the MDBS access control mechanism must make a choice in this case for each of the three unspecified access operations - update, delete and insert. A naive access control mechanism may permit user 1 to update, delete or insert records containing keywords derived from the descriptor ($0 < \text{Salary} \leq 100$), since such operations have not been specifically disallowed in the field-level access controls by the database creator. However, such naive access control would permit user 1 to compromise the access control of the database. We illustrate the problem of compromised access control with the following example.

A. A Case of Compromised Access Control Due to Alternative Operation

Let us consider a salary database in which a typical and single-attribute-valued record R1 is

R1: (<Salary,50>).

Let the database creator specify the following descriptor and corresponding field-level access control for user 1.

($0 < \text{Salary} \leq 100$) ([Salary],No-Retrieve)

Let cluster 1 consist of all the records whose keywords are derived from the descriptor ($0 < \text{Salary} \leq 100$). Then, it follows that the set of field-level access controls for user 1 corresponding to cluster 1 is

([Salary],No-Retrieve)

That is, user 1 is not permitted to retrieve the salary field from records belonging to cluster 1. However, there is no indication as to whether user 1 is permitted to update or delete (or insert) the (a) salary field from (to) records belonging to cluster 1. Let us assume that a naive access control mechanism permits user 1 to update the salary field of records belonging to cluster 1. For instance, the following update request by user 1 is allowed to take place.

UPDATE (Salary = 50) <Salary = Salary + 0>

If the salary database does not contain a record for an employee earning 50, the mechanism will return a negative acknowledgement to the user. On the other hand, if the database does contain a record for an employee earning 50, the mechanism will return a positive acknowledgement to the user. Thus, user 1 can easily make the inference whether there is an employee record in the database with earning being 50. In other words, user 1 may infer at the information which he would have obtained if he had issued the retrieve request

RETRIEVE (Salary = 50) (Salary)

Since user 1 is not allowed to issue the above retrieve request as dictated by the field-level access controls specified by the database creator, the user obtains the salary information of the employee by way of an update request. Any access control mechanism which allows information to be revealed to a user by way of one type of requests while the same information is expressly prohibited from being revealed to the user under the other types of requests is said to suffer from the problem of compromised access control as characterized in the above case study. Obviously, the MDBS access control mechanism must not have the problem of compromised access control.

In the example, the problem of compromised access control arises because user 1 is allowed to update the value of an attribute from records in a specific cluster, even though he is not allowed to retrieve the value of this attribute from records in the cluster. Clearly, the problem may be overcome by a more sophisticated access control mechanism which ensures the following rule. Whenever a user is not allowed to retrieve the value of an attribute from records in a cluster, the user must also not be allowed to update the value of that attribute from records in the cluster. We term such a rule, the rule of access control implication. We therefore say that the No-Update disallowed access operation is implied by the No-Retrieve disallowed access operation. Graphically, we denote the implication as follows.

$$\text{No-Retrieve} \rightarrow \text{No-Update}$$

For each user, the MDBS access control mechanism will enforce not only the set of field-level access controls over every cluster induced by the database creator's specification, but also all the other disallowed access operations implied by disallowed access operations in the field-level access controls.

B. Two More Cases of Compromised Access Control Due to Alternative Operation

In addition to the case of compromised retrieval due to update, there are other implications. Disallowed update operations may be compromised by delete operations, since No-Delete is implied by No-update. The converse is also true. Consider, once again, the salary database of the previous example. Let us assume that the database creator specifies the following descriptor and corresponding field-level access control for user 1.

$$(0 < \text{Salary} \le 100) \quad ([\text{Salary}],\text{No-Update})$$

Furthermore, let cluster 1 consist of all the records whose keywords are derived from the descriptor $(0 < \text{Salary} \le 100)$. Then, it follows that the set of field-level access control for user 1 corresponding to cluster 1 is

([Salary],No-Update).

That is, user 1 is not allowed to update the salary field of the records belonging to cluster 1. However, there is no indication of whether user 1 is allowed to retrieve the salary field from records belonging to cluster 1. Nor is there any indication of whether user 1 is allowed to insert new salary into or delete old salary from such records. To illustrate how a naive access mechanism may compromise the database by allowing user 1 to insert and delete records of cluster 1, we propose the following scenerio.

Knowing that update operation is denied from the user, user 1 tries the following requests in sequence.

DELETE (Salary = 50)
INSERT (<Salary,52>)

Thus, user 1 is able to achieve the effect of updating the salary by 2. The access control is compromised because the user is disallowed to update any salary of records in cluster 1, and yet the user is able to achieve the effect of update by means of delete and insert operations. Clearly, the problem of compromised access control in the example may be overcome by either disallowing the insert operation or the delete operation on records in cluster 1. In general, a sophisticated access control mechanism should recognize the following implications

$$\text{No-Update} \rightarrow \text{No-Delete},$$
$$\text{No-Update} \rightarrow \text{No-Insert}$$

Conversely, we learn that if a user is not allowed to delete records from a cluster, the user must not be allowed to update attribute values of records in the cluster. That is, the implication

$$\text{No-Delete} \rightarrow \text{No-Update}$$

holds. To illustrate the necessity in upholding the implication, we present the following counterexample. Assume that the field-level access control corresponding to cluster 1 for user 1 is

([ ],No-Delete),

where the '[ ]' indicates that the attribute combination part is unspecified. Since the update opeartion on records in cluster 1 has not been specifically disallowed in the field-level access control, a naive access control mechanism may allow user 1 to update all the fields of records in cluster 1. Thus, user 1 may achieve the effect of deleting a record by updating the value of every field in the record to null. Hence, the problem of compromised access control exists.

We conclude that in order to overcome the problem of compromized access control, a sophisticated access control mechamism must enforce the following four implications:

(1) No-Retrieve $\rightarrow$ No-Update
(2) No-Retrieve $\rightarrow$ No-Delete
(3) No-Update $\rightarrow$ No-Delete
(4) No-Delete $\rightarrow$ No-Update

Now, we are ready to describe the process of request execution with access control for non-insert requests. We have already described, in Section 4.1.3, how descriptor processing and the first step of address generation are used to determine the set of clusters corresponding to the query in the user request. Now, we need to do the following for each cluster k in this set in order to exercise access control. By looking up an entry in the augmented CDT corresponding to the row for cluster k and the column for the user id of the user that issued the non-insert request, each back-end will make the following checks.

If the request is a delete request, each back-end will check to see if the entry of the augmented CDT contains a field-level access control with any attribute combination part and with 'No-Delete', 'No-Retrieve' or 'No-Update' in the disallowed access part. By checking for 'No-Retrieve' in the disallowed access part, the implication

No-Retrieve $\rightarrow$ No-Delete

is being upheld.

Similarly, by checking for 'No-Update' in the disallowed access part, the implication

$$\text{No-Update} \rightarrow \text{No-Delete}$$

is being upheld. Since both the implications related to the delete request are thus upheld by each back-end, the problem of compromised access control does not occur in MDBS during the execution of the delete request.

If the request is an update request, each back-end will check to see if the entry of the augmented CDT contains a field-level access control with the attribute being modified as one of the attributes in the attribute combination part and the disallowed access part as 'No-Update' or 'No-Retrieve'. For instance, if the attribute being modified is salary, it will check to see if the entry contains a field-level access control of the form

([Salary,...],No-Update) or ([Salary,...],No-Retrieve),

where the attribute combination part may have other attributes besides salary. By checking for 'No-Retrieve' in the disallowed access part, the implication

$$\text{No-Retrieve} \rightarrow \text{No-Update}$$

is being upheld. Each back-end will also check to see if the entry contains a field-level access with an unspecified attribute combination part and with 'No-Delete' in the disallowed access part. This allows the back-end to uphold the implication

$$\text{No-Delete} \rightarrow \text{No-Update}$$

Since both the implications related to the update request are thus upheld by each back-end, the problem of compromised access control cannot occur in MDBS during the execution of the update request.

If the request is a retrieve request, each back-end will check to see if the entry of the augmented CDT contains a field-level access with the attribute combination part containing one or more attributes from the target-list (of the retrieve request) and the disallowed access part being 'No-Retrieve'. For example, consider the retrieve request

RETRIEVE (File = EMPLOYEE) (Name,Salary).

In this case, each back-end will check to see if the entry contains a field-level access control of the form

([Name,...],No-Retrieve) or ([Salary,...],No-Retrieve).

Here, we have only described the processing of retrieve requests which do not contain aggregate operators in their target-lists. The processing of requests with aggregate operators is described later in Section 5.4.

If the check is positive in any of the cases mentioned above, the back-ends delete cluster k from the set of corresponding clusters. In other words, cluster k is a cluster relevant to the user request; nevertheless, it is not authorized for the user. The above procedure is repeated for every cluster in the set of corresponding clusters. The remaining set of clusters is referred to as the <u>permitted</u> or <u>authorized</u> <u>set</u> <u>of</u> <u>clusters</u> for the non-insert request. Once the set of permitted clusters for a non-insert request have been determined, the back-ends proceed with the third step of address generation and with record processing as described in Section 4.3.

5.3.3 The 'Conservative' and Precision Access Control Mechanism

First, we would like to point out that the MDBS access control mechanism may be 'conservative' in that it may reject a user request by 'overly' protecting the requested data. For instance, consider the following retrieve request issued by user 1.

RETRIEVE (Dept = TOY) (Salary)

For simplicity, assume that all the relevant records of the toy department belong to cluster 1 and that the set of field-level access control for user 1 corresponding to cluster 1 has been determined as

([Name,Salary],No-Retrieve).

In this case, Salary is the only attribute in the target-list of the retrieve request. Also, Salary is one of the attributes in the attribute combination part of the field-level access control for user 1 corresponding to cluster 1. Thus, according to the algorithm presented in the previous section, the access to the salary field is denied to user 1 and the request is rejected. However, it might be the intention of the database creator that user 1 is only to be disallowed from reading the name and salary fields jointly from records in cluster 1 in order to prevent the user from learning individual employee salaries. That is, the database creator might intend to allow user 1 to access either the name or salary fields singly but not jointly from records in cluster 1, since separate lists of names and salaries might not correlate the individual employee salaries. In view of this example, MDBS access control mechanism might be considered to be conservative in not allowing user 1 to access the salaries from such records.

However, there is a very good reason for the conservatism of the MDBS access control mechanism. Consider that, in the above example, user 1 were allowed to read the name and salary fields individually from records in cluster 1. Then, user 1 may issue the following two requests one after the other.

RETRIEVE (Dept = TOY ∧ SS# = 50) (Name), and
RETRIEVE (Dept = TOY ∧ SS# = 50) (Salary),

As a result, the user may obtain the name and salary of the employee in the toy department where social security number is 50. But this may be precisely the information that the database creator may intend to disallow user 1 from obtaining. In other words, if the MDBS access control mechanism is not 'conservative', it is possible for users to compromise the access control of the database. It is to overcome the problem of compromised access control that MDBS has a 'conservative' access control mechanism.

Second, we note that MDBS access control mechanism may eliminate one or more clusters from consideration in the second step of address generation. Since a cluster is stored in one or more tracks, the elimination of a cluster from consideration will result in savings in terms of reduced number of accesses to the secondary store.

We also note that in MDBS all access control checks are made prior to the retrieval of records from the secondary store. Consequently, access imprecision due to redundant record retrieval never occurs. Thus, the need to discard retrieved records due to access control violations does not arise since they are retrieved from the secondary memory only after they are cleared for access.

5.3.4 An Example of the Process of Request Execution With Access Control

In order to illustrate the entire process of request execution with access control, we consider the database of Figure 33 and the corresponding augmented CDT of Figure 37. Let us assume that the user identified as user 2 issues the request

RETRIEVE (Salary < 10000) (Manager).

This request requires the retrieval of the names of managers of all those employees who earn less than $10,000. After descriptor processing and the first step of address generation, MDBS determines that the corresponding set of clusters for the query (Salary < 10000) is the set consisting of cluster 1 and cluster 2. In the second step of address generation, the access control takes place. In this step, the permitted set of clusters must be determined. The two clusters in the set of corresponding clusters are checked, in turn. Cluster 1 is checked for authorization by looking up the entry corresponding to cluster 1 and user 2 in Figure 37. There is the following entry

(([Manager],No-Update), ([Employee,Salary], No-Retrieve))

which is checked to see if it contains a field-level access control of the form ([Manager, ...], No-Retrieve). Since no such field-level access control is present in the entry for cluster 1, cluster 1 is a permitted cluster. Next, cluster 2 is checked for authorization by looking up the entry corresponding to cluster 2 and user 2. There is the following entry

(([Manager],No-Retrieve), ([Employee,Salary], No-Retrieve)).

which is checked to see if it contains a field-level access control of the form ([Manager, ...],No-Retrieve). Since such a field-level access is indeed present in the entry for cluster 2, cluster 2 is not a permitted cluster. Thus, cluster 1 is the only permitted cluster. Then, the third step of address generation and record processing are performed for the retrieve request. Looking at Figure 36 again, we see that records R1, R2 and R4 are retrieved, since these are the only records in cluster 1. Finally, the value of the Manager attribute is retrieved from these records and presented to user 2. This completes the processing of the retrieve request for user 2.

5.4 New Capabilities of the Access Control Mechanism

In this section, we describe two extensions to the basic access control mechanism. These optional extensions are discussed in turn.

5.4.1 Statistical Access Control

It may be necessary to disallow a user from obtaining an aggregate value of some attribute over a number of records. Aggregate attribute values are resulted from the use of aggregate functions such as average, summation, maximum and minimum. Our access control mechanism may be extended to provide such a capability as described below.

We recall that a field-level access control is a pair of the form (attribute combination, disallowed access operation), where an attribute combination is a set of attributes. We now extend our definition of attribute combination to be a set of elements. Each element is either an attribute, e.g., Salary or an aggregate operator to be performed on an attribute, e.g., AVG(Salary). An aggregate operator is one of AVG, SUM, COUNT, MAX, MIN. An example of an attribute combination with two elements is

[Department, AVG(Salary)].

Let us now demonstrate how attribute combinations with aggregate operators may be used to control access over requests for aggregate values. Consider, for example, an employee database in which the database creator wishes to disallow user 1 from obtaining the average salary of all those employees who are working in the toy department. However, the database creator allows the same user to obtain the average salary of all those employees who are working in the sales department. Finally, the database creator disallows user 1 from obtaining the individual salaries of employees in either of these two departments. Then, the database creator may specify the following two descriptors and corresponding sets of field-level access controls.

(Department = TOY)	([AVG(Salary),Salary],No-Retrieve)
(Department = SALES)	([Salary],No-Retrieve)

For simplicity, let cluster 1 be defined by the single descriptor

(Department = TOY).

Also, let cluster 2 be defined by the single descriptor

(Department = SALES).

Then, it is easily determined that the set of field-level access controls for user 1 corresponding to cluster 1 is

([AVG(Salary),Salary],No-Retrieve).

Similarly, the set of field-level access controls for user 1 corresponding to cluster 2 is determined as

([Salary],No-Retrieve).

Now, assume that user 1 issues the request

RETRIEVE (Department = TOY) (AVG(Salary)).

MDBS will first perform descriptor processing and the first step of address generation to determine that the only cluster corresponding to the query in the user request is cluster 1. Since the set of field-level access controls for user 1 corresponding to cluster 1 disallows access to the salarv average, the request from user 1 is rejected. On the other hand, assume that user 1 issues the request

RETRIEVE (Department = SALES) (AVG(Salary)).

In this case, MDBS will first perform descriptor processing and the first step of address generation to determine that the only cluster corresponding to the query in the user request is cluster 2. Since the set of field-level access control for user 1 corresponding to cluster 2 does not disallow access to the salary average, the request is not rejected. MDBS will determine the secondary memory addiesses of the records in cluster and retrieve these records. It will then extract the values of the salary attribute from such records , compute the average and output a single value for the average salary. This average salarv will then be returned to user 1.

Now, consider that user 1 issues the request

RETRIEVE (Department = TOY) ∨ (Department = SALES) (AVG(Salary)).

In this case, MDBS will perform descriptor processing and the first step of address generation to determine that the set of clusters corresponding to the query in the request is the set consisting of clusters 1 and 2. In the second step of address generation, MDBS will determine the set of permitted clusters. Since the set of field-level access controls for user 1 corresponding to cluster 1 does not allow access to the salary average, cluster 2 is the only permitted cluster. Now, MDBS may take one of two possible courses of action. One possibility is to access the records in cluster 2, extract the values of the salary attribute from such records, compute a salary average and return this average salary to the user. However, the average salary returned to user 1 is restricted to the employees in the sales department. What the user had requested was the average salary of all employees in both the sales and toy departments. Therefore, MDBS must also return a message to user 1 indicating that the value returned is only an authorized one. The second possible course of action is simply to reject the request, since the requested average touches upon unauthorized data. Either course of action may be chosen for our implementation.

A. An Example of Compromised Statistical Access Control Due to Users' Own Aggregate Operations

Consider a salary database and let the database creator specify the following descriptor and corresponding set of field-level access control for user 1.

$$(0 < \text{Salary} \le 100) \quad ([\text{AVG(Salary)}],\text{No-Retrieve})$$

That is, the database creator specifies that user 1 is not allowed to access the salary average of all records containing keywords derived from the descriptor

$$(0 < \text{Salary} \le 100).$$

Let us assume that cluster 1 is defined by this same descriptor. Then, it is easy to see that the set of field-level access control for user 1 corresponding to cluster 1 is

$$([\text{AVG(Salary)}],\text{No-Retrieve}).$$

That is, user 1 is not allowed to access the salary average of the records in cluster 1. However, there is no indication of whether the user is allowed to access the individual salary field of such records. Consider a naive access control mechanism which allows user 1 to access the salary field from records in cluster 1. Now, user 1 may access the salary field from each and every record in cluster 1. Then, the user may compute the average of all these retrieved salaries (either manually or by running a statistical program). In this way, user 1 may obtain the average salary of all records in cluster 1, thus compromising the access control of the database. Therefore, a sophisticated access control mechanism will not permit user 1 to access the salary field from records in cluster 1.

In order to ensure that the statistical security of the database is not compromised, MDBS access control mechanism incorporates the following additional procedure in the process of request execution with access control for retrieve requests.

B. An Expanded Procedure for Effective Statistical Access Control

We recall that the attribute combination part of a field-level access control and the target-list of a retrieve request are both lists of elements where each element is either an attribute or an aggregate operator to be performed on an attribute. Whenever an element in an attribute combination part of a field-level access control is an aggregate operator on an attribute, the attribute is also added to the attribute combination part of the field-level access control. Thus, for example, the field-level access control

([AVG(Salary),Dept],No-Retrieve),

becomes in effect

([AVG(Salary),Salary,Dept],No-Retrieve).

Every field-level access control in the relevant entry of the augmented CDT is expanded in this way. Now, each back-end will check to see if the entry of the augmented CDT contains a field-level access control with the attribute combination part containing one or more elements from the target-list of the retrieve request. If the check is positive, each back-end will delete cluster k from the set of corresponding clusters. The above procedure is repeated for every cluster in the set of corresponding clusters. Thus, we form the set of permitted clusters for the retrieve request. Then, each back-end may proceed with the final step of address generation and with record processing as described in Section 4.3.

5.4.2 Value-Dependent Access Control

In this section, we will describe how MDBS may control access to data on the basis of the relationship of the attribute value of the data and the user of the data. In this case, the access control is dependent on the values of a specific attribute. Thus, we refer to it as value-dependent access control. Furthermore, we shall refer to the specific attribute whose values are used to enforce access control as the relationship attribute. Let us illustrate the value-dependent access control with an example.

Consider that the database creator wishes to give a user access to the records of those and only those employees who are managed by the user. In this case, the relationship attribute is Manager. With respect to the sample database of Figure 33, employee 1 may only access the first two records, employee 3 may only access the next two records, and employee 5 may only access the last two records.

In this case, value-dependent access control is implemented in MDBS as follows. As before, the database creator specifies a number of descriptors and corresponding sets of field-level access controls at the database-creation time. In addition, the database creator specifies a relationship attribute and a set of field-level access controls corresponding to that attribute. This single relationship attribute and single set of field-level access controls must be converted by the system into a number of descriptors and corresponding sets of field-level access controls. For simplicity, we will select from the descriptors specified by the database creator at the database-creation time the only relationship attribute and the corresponding set of field-level access controls for consideration. In our example, we want to disallow a user all accesses to records of employees whom the user does not manage. That is, the only relationship attribute is Manager and the set of field-level access controls is the one which disallows all accesses. In the sequel, we shall designate this set of field-level access controls with 'All'.

MDBS must convert the relationship attribute and set of field-level access controls to a number of descriptors and p sets of field-level access controls for each descriptor, where p is the number of potential users of the database. Once this conversion is achieved, the algorithm described in Section 5.2 may be used to form the augmented cluster definition table. Then, this table may be used as before during request execution with access control. In the next paragraph, we explain how the relationship attribute and set of field-level access controls are converted to a number of descriptors and p sets of field-level access controls.

The conversion is illustrated with the use of the sample database of Figure 33. We assume that each employee in the database of Figure 33 is permitted to access the database and that these six employees are the only users of the database. Thus, p is set to six. The conversion algorithm is as follows. First, we extract all possible values of the relationship attribute from the records of the database and place it in a set V. In our example, we will extract all possible values of the Manager attribute from the records of Figure 33 and form the set V as {1,3,5}. For each value v in the set V, we will form a descriptor of the form

(relationship attribute = v)

In our example, we will form the three descriptors (Manager = 1), (Manager = 3) and (Manager = 5). Corresponding to each descriptor so formed, we need to form p sets of field-level access controls. In our example, corresponding to each of the three descriptors, we need to form six sets of field-level access controls. The set of field-level access controls corresponding to user i for a descriptor of the form (relationship atrribute = v) is set to 'All' (the database creator specified set of field-level access control) if i is not equal to v. Otherwise, it is set to null to indicate that no access is disallowed. The descriptors and corresponding sets of field-level access controls formed for our example are shown in Figure 38. This completes our description of the algorithm for converting from the database-creator-specified relationship attribute and corresponding set of field-level access controls to a number of descriptors and p sets of field-level access controls, where p is the number of potential users of the database.

Once the descriptors and corresponding sets of field-level access controls are obtained, the algorithm described in Section 5.2 may be used to create the augmented cluster definition table and the algorithm of Section 5.3 is used to enforce access control during request execution.

5.5 Access Control for Transactions of Multiple Requests

Up to this point, we have described the process of request execution with access control for individual requests. However, as described in Chapter 3, we also allow for transactions of multiple requests to be carried out in MDBS. In this section, we consider the process of transaction execution with access control. Two techniques are considered for executing transactions consisting of of multiple requests. The first technique, called the stand-alone execution is known for its simplicity. The second technique, called attached execution is more complex. Nevertheless, it is more effective and efficient in controlling access of transactions to the database.

5.5.1 Stand-Alone Execution of Transactions

Each request in a transaction is executed in the manner described in Section 5.3. Thus, no special attention is paid to the fact that these requests are actually parts of a transaction.

5.5.2 Attached Execution of Transactions

Each request in a transaction is executed till the point where the set of authorized clusters for the request are determined. That is, descriptor processing and the first two steps of address generation are performed on each request in the transaction. Now, each request in the transaction may be

Descriptor	Set of Field-Level Access Controls for User 1	Set of Field-Level Access Controls For User 2	Set of Field-Level Access Controls For User 3	Set of Field-Level Access Controls For User 4	Set of Field-Level Access Controls For User 5	Set of Field-Level Access Controls For User 6
(Manager=1)	–	All	All	All	All	All
(Manager=3)	All	All	–	All	All	All
(Manager=5)	All	All	All	All	–	All

'All' indicates that all accesses are *disallowed*

'–' indicates that no access is disallowed

Figure 38. The Descriptor and Corresponding Sets of Field-Level Access Controls Formed for the Database of Figure 33

attached with the authorized set of clusters for that request. We shall refer to an attached request as a secure request because access control processing has been completed for this request. By forming a secure request from every request in a transaction, we may form a secure transaction from the original transaction. The secure transaction formed from the transaction is referred to as the secure version of the transaction. Note that each request in a secure transaction has only been partially but not completely executed. After a secure transaction has been formed, each request in the secure transaction may then be executed to its completion by executing the third step of address generation followed by record processing.

5.5.3 The Choice of Transaction Execution for Access Control

There are two reasons that we prefer to select the attached execution technique for controlling access of transactions to the database. In order to explain the first reason, we define the permitted set of clusters for a transaction as the union of the authorized set of clusters for each request in the transaction. In Chapter 6, we will see that, to enforce concurrency control, a transaction must lock its set of permitted clusters before any access to the secondary memory may be made on behalf of the transaction. Therefore, it is necessary to preprocess transactions to determine their sets of permitted clusters. This is precisely what we do in the attached execution technique described above. On the other hand, in the stand-alone execution technique, a transaction may be partially executed even before its set of permitted clusters is entirely determined. If we were to employ the stand-alone execution technique, then no lock for concurrency control could be placed on the permitted set of clusters for a transaction since such a set is not known before hand.

The second reason for preferring the attached execution technique has to do with the fact that transactions in MDBS are meant to be executed repeatedly. If this technique is employed, we need to enforce access control for the requests in a transaction only the first time the transaction is executed. After this first execution of a transaction, a secure version of it may be preserved for use during subsequent executions of this transaction. On the other hand, if the stand-alone execution technique is employed, it is necessary to enforce access control each time a transaction is executed, since a secure version of the transaction is not available.

It should be emphasized, nevertheless, that in the attached execution technique the preserved secure version of a transaction may be used in subsequent executions of that transaction only as long as the database creator has not made any changes to the access control information about the database. If the database creator does make modifications to the access control information, the preserved secure versions of transactions are no longer valid and must be recalculated.

5.6 The Management of Access Control Information

From time to time, a database creator may change the access control information for his or her database. We shall consider the various ways in which a database creator may change the access control information and the various actions to be taken by MDBS in response to these changes. Here, we find it necessary to talk at the implementation level rather than at the logical level of the access control information. Accordingly, we keep in mind that in addition to the augmented descriptor-to-descriptor-id table (DDIT), there is the augmented CDT which is physically implemented as one cluster definition table (CDT) and separate cluster definition tables (SCDTs), one for each user of the database. Let us now proceed to discuss the various changes that a database creator may wish to make to the access control information.

5.6.1 Denying a User From Any Access to the Database

Consider that the database creator wishes to disallow one of the users from accessing the database. In that case, each back-end will simply delete the SCDT for that user. Thus, the actions taken by MDBS in response to this change made by the database creator are simple.

5.6.2 Adding a New User of the Database

Let us consider that the database creator wishes to add a new user to the system. In this case, he must first acquire a new user id from MDBS which the database creator will pass on to the new user. Next, the database creator needs to specify one set of field-level access controls for each descriptor in the database. These sets of field-level access controls are stored as a new column of the augmented DDIT. Pictorially, the augmented DDIT of Figure 34, before and after the addition of a new user (i.e., user 3) is shown in Figure 39. In the figure, F31, F32, F33 and F34 are the sets of field-level access controls

Original from Figure 33

Descriptor Id	Descriptor	Set of Field-Level Access Controls For User 1	Set of Field-Level Access Controls For User 2
D1	(1000 ≤ Salary ≤ 10000)	F_{11}	F_{21}
⋮	⋮	⋮	⋮
D4	(Department = 3)	F_{14}	F_{24}

New Augmented DDIT Due to Changes of Above

Descriptor Id	Descriptor	Set of Field-Level Access Controls For User 1	Set of Field-Level Access Controls For User 2	Set of Field-Level Access Controls For User 3
D1	(1000 ≤ Salary ≤ 10000)	F_{11}	F_{21}	F_{31}
⋮	⋮	⋮	⋮	⋮
D4	(Department = 3)	F_{14}	F_{24}	F_{34}

F_{ij} : Set of Field-level Access Controls for User i Corresponding to Descriptor j

Figure 39. The Augmented DDIT of Figure 33 Before and After the Addition of User 3 to the System

specified for the new user. After a new column has been added to the augmented DDIT, each back-end must next form an SCDT for the new user. This requires the back-end to determine the sets of field-level access controls for user 3 on every cluster. The algorithm described in Section 5.2 is used to do this. Once the SCDT for user 3 is created, all requests issued by user 3 may be executed in the manner described in Section 5.3.

5.6.3 Changing the User's Access Operations Only

The database creator may wish to change the corresponding set of field-level access controls for a user on a descriptor. Then, MDBS performs the following operations. First, it makes the appropriate changes to the augmented DDIT. For instance, let us assume that we wish to change the set of field-level access controls for user 2 on descriptor D1 from ([Employee, Salary],No-Retrieve) as shown in Figure 34 to ([Employee],No-Retrieve). The augmented DDIT of Fig. 34, after appropriate changes have been made to it, is now shown in Figure 40. After appropriate changes have been made to the augmented DDIT, each back-end will then modify the SCDT of the user whose set of field-level access controls was changed by the database creator. In general, the new sets of field-level access controls for the user correspond only to some of the clusters. Thus, only the access control information of these clusters needs to be modified in the user's SCDT. Specifically, in our example, the sets of field-level access controls, for user 2 corresponding to all those clusters whose definitions contain D1, need only to be modified. This is done by recalculating the sets of field-level access controls for these clusters using the algorithm of Section 5.2. The original SCDT, before modification, for user 2 is the one shown in Figure 36. The modified SCDT for user 2 is shown in Figure 41. Once the SCDT for user 2 is modified, all requests issued by user 2 may be executed in the manner described in Section 5.3.

5.6.4 Changing the User's Permitted Data Grandules

There are four other changes which a database creator may wish to make. A database creator may wish to add a new descriptor, delete an existing descriptor, coalesce two existing descriptors into a single descriptor, or split up an existing descriptor into two descriptors. Since descriptors are used to define clusters, changes in descriptors will cause changes in cluster definitions and in the sets of field-level access controls corresponding to clusters.

Descriptor Id	Descriptor	Set of Field-level Access Controls For User 1	Set of Field-level Access Controls For User 2
D1	$(1000 \leq \text{Salary} \leq 10000)$	—	([Employee],No-Retrieve)
D2	$(10000 < \text{Salary} < \infty)$	—	All
D3	$(1 \leq \text{Department} \leq 2)$	—	(([Manager],No-Retrieve), ([Employee,Salary], No-Retrieve))
D4	$(\text{Department} = 3)$	—	([Manager],No-Retrieve)

'-'	indicates that no access is disallowed
'All'	indicates that all accesses are disallowed

Figure 40. The Augmented DDIT After Changing The Set of Field-Level Access Controls for User 2 on Descriptor D1

(See Figure 36 for the original SCDT for User 2)

Cluster id	Set of Descriptors	Records in the Cluster Defined by the Descriptor Set	Set of Field-level Access Controls
1	{D1,D3}	R1,R2,R4	(([Manager],No-Update),([Employee],No-Retrieve))
2	{D1,D4}	R5	(([Manager],No-Retrieve),([Employee],No-Retrieve))

Figure 41. The New Separate Cluster Definition Table (SCDT) for User 2

Thus, it will be necessary to delete the existing SCDTs and to create new SCDTs. Thus, each of these changes on the part of the database creator will require some work on the part of MDBS.

A. Defining New Granules With New Descriptors

Consider, first, the addition of a new descriptor. In this case, the database creator also needs to specify p sets of field-level access controls, one set for each user allowed to access the database. Now, a row for this new descriptor may be added to the augmented DDIT. Then, the algorithm presented in Section 5.2 is used to determine the sets of field-level access controls for every user on every cluster. In this way, several new SCDTs are created at each back-end, one for every user allowed to access the database. This completes the actions to be taken by MDBS in response to the addition of a new descriptor by the database creator.

B. Creating Larger Granules By Deleting Descriptors

Let us consider what happens when one of the existing descriptors is deleted from MDBS. First, the row corresponding to this descriptor will be deleted from the augmented DDIT. Then, the algorithm presented in Section 5.2 is used to determine the sets of field-level access controls for every user on every cluster. Thus, several new SCDTs are created at each back-end to replace the deleted SCDTs. Thus, the actions to be taken by MDBS in this case are very similar to the actions taken by MDBS when a new descriptor is added.

C. Creating Larger Granules by Coalescing Descriptors

Let us consider that the database creator wishes to coalesce two descriptors into a single descriptor. For instance, he may wish to coalesce the descriptor ($0 < \text{Salary} \leq 100$) and the descriptor ($101 < \text{Salary} < 200$) into the single descriptor ($0 < \text{Salary} < 200$). In this case, the database creator also needs to specify p sets of field-level access controls for the new single descriptor, one set for each user allowed to access the database. MDBS responds by first appropriately modifying the augmented DDIT. This modification consists of the removal of the rows corresponding to the two descriptors being coalesced and the addition of a row for the new single descriptor. Then, the algorithm of Section 5.2 may be used by each back-end to create the new SCDTs of the database to replace the deleted ones.

D. Defining New Granules By Splitting Descriptors

The final case we consider is when a descriptor needs to be split up into two descriptors. Consider that the database creator wishes to split descriptor D into descriptors D1 and D2. This may be considered as the deletion of descriptor D followed by the addition of descriptors D1 and D2. Since the actions to be performed by MDBS during descriptor addition and descriptor deletion have already been specified, the actions to be performed by MDBS during descriptor splitting is also specified.

6. CONCURRENCY CONTROL FOR MULTIPLE BACK-ENDS AND CONSISTENCY OF PARTITIONED DATABASES

In this chapter, we will first raise the issue whether MDBS indeed needs any form of concurrency control. Once we are convinced that we need some form of concurrency control, we will then examine the various existing concurrency control mechanisms in terms of MDBS requirements. In the examination, we discover that a new requirement for consistency, known as monolithic consistency, will have to be met. We will therefore strive for a simple concurrency control mechanism for upholding the monolithic consistency in MDBS. This chapter will mostly be devoted to the design and illustration of the MDBS concurrency control mechanism.

6.1 Is there a Necessity for Concurrency Control?

In a multiple back-end database system, there is no need for concurrency control as long as each back-end is neither executing multiple requests in a concurrent fashion nor executing requests from multiple users in an interleaved fashion. Let us elaborate this observation with the following discussion. We recall that a transaction consists of multiple requests. Let transaction t1 consist of requests r1 and r2, and let transaction t2 consist of requests r3 and r4. If a back-end executes r3, after r1 has been completely executed, then we say that the back-end is executing requests (i.e., transactions) from multiple users in an interleaved manner. We note that in this case there is no concurrent execution of requests. A back-end is said to be executing two requests in a concurrent manner if both requests have been partially executed by the back-end, but neither request has been completely executed. Is it necessary for a back-end to be executing requests in a concurrent and in an interleaved fashion?

The argument against interleaved execution of requests from multiple users and against concurrent execution of multiple requests for the same or different user is that it obviates the need for having a concurrency control mechanism at each back-end, thereby making the software at each back-end simpler. The designer of the distributed database system, known as the Stonebraker's Machine [Ston78], supports the argument with additional points. It is felt that the back-ends should not support concurrent request execution, because concurrent request execution would not be beneficial to the system performance when only one disk drive is attached to a back-end. In [Ston78], it was also argued that concurrent request execution will prove advantageous only if multiple disk drives are connected to each back-end.

In view of the above cases of points, we argue strongly that MDBS should support concurrent and interleaved execution of user requests and therefore concurrency control for the following reasons.

A. The Throughput Issue

The disadvantage of not allowing for concurrent request execution is that the throughput of MDBS will be poorer. This is because database requests in MDBS are mostly I/O-bound. Thus, a back-end will be mostly waiting for the disk to access and retrieve data. The use of clusters and placement strategy in our system ensures us that only a small amount of processing will be needed after the data is retrieved. Thus, the idle time is likely to be high in a back-end unless the back-end utilizes the time for the execution of concurrent requests.

B. The Response-Time Issue

We also believe that interleaved execution of user requests should also be supported at the back-ends of MDBS. The reasons for supporting interleaved execution of user requests is that, otherwise, the response time to a user's request will be high.

C. The Multiple Disk Drive Issue

There are two good reasons for attaching multiple disk drives to a minicomputer back-end. First, minicomputers are powerful enough to handle multiple disk drives. For example, a PDP-11/44 is designed to operate with up to 64 disk drives. If a back-end were to be attached to only a single disk drive, it would appear to be functioning like a disk controller. In fact, even a conventional disk controller is capable of managing multiple disk drives. Thus, we would be making poor use of the processing power of a minicomputer if we were to attach only a single disk drive to it. The second reason for having more than one disk drive at each back-end is that we are trying to support very large databases. We cannot support very large databases by attaching only one disk drive to a back-end.

From the above arguments, we conclude that MDBS should support both concurrent and interleaved execution of user requests. We therefore endeavor to to devise a concurrency control mechanism for such support.

6.2 What are the Necessary and Sufficient Conditions for a Consistent Partitioned Database Utilizing Multiple Back-ends?

Concurrency control mechanisms are developed for the purpose of upholding the consistency of the database. For example, in centralized databases, concurrent control mechanisms must ensure inter- and intra-consistencies of data [Hsia81]. In distributed databases, on the other hand, concurrency control mechanisms must ensure, in addition to inter- and intra-consistencies, mutual consistency [Thom79]. We first argue that mutual consistency of multiple copies of the same data is not necessary for a partitioned database such as ours. We note that only the augmented descriptor-to-descriptor-id-table (DDIT) is duplicated in MDBS. In other words, the same augmented DDIT is stored at each back-end of MDBS, where as the augmented cluster definition tables (CDTs) and the database are not duplicated at the back-ends. Only the distinct augmented CDT and different clusters of the database are stored at individual back-ends. Furthermore, update requests issued by a user will never cause any modification of the augmented DDIT. The remaining data which may be modified by the update requests are stored as a partitioned database and requires no copy of the same data. We also argue with the following counter-example that inter-consistency of data is not sufficient for a partitioned database. Thus, the necessary and sufficient conditions for a consistent database in a multiple back-end system will not be found in the conventional solutions to centralized and distributed database systems. We must identify the new consistency problem for partitioned database systems.

Let us show, by means of examples, the exact nature of the problem in the context of MDBS. Consider that the following two updates are issued to MDBS.

```
UPDATE (File = EMPLOYEE) <Salary = Salary + 2>
UPDATE (File = EMPLOYEE) <Salary = Salary * 2>
```

The first request increments the salaries of all employees by two dollars. The second request doubles the salaries of all employees. Clearly, if we allow a back-end to concurrently execute these two requests, inconsistent results may arise. Thus, some employees will have their salaries doubled; others will have their salaries incremented by two dollars; others may have their salaries changed from an original value, say x, to (2x + 2); and still others may have their salaries changed from an initial value x to a final value 2(x + 2). These inconsistencies go by various names such as the problem of lost updates and the non-reproducibility of reads [Gard77]. Two requests, such as those above, whose simultaneous (or concurrent) execution can lead to inconsistencies are termed incompatible requests. More formally, we say that two requests r1 and r2 are compatible if their simultaneous execution gives the same result as would be obtained if their execution sequence is r1 followed by r2 or r2 followed by r1. In order to ensure that such inconsistencies do not occur, a concurrency control mechanism must ensure that two incompatible requests are executed one after the other. A mechanism which ensures serial execution of incompatible requests is said to maintain inter-consistency [Hsia81].

It is clear that if each back-end is to execute requests, one after the other, instead of concurrently, no inconsistency can result. When we generalize from single requests to transactions of multiple requests, then it is clear that if each back-end is to execute transactions one after the other and also execute all the requests within a transaction one after the other, no inconsistency can result. Since any concurrency control mechanism for a centralized database system ensures inter-consistency, one possible approach would be to have a centralized concurrency control mechanism in the controller of MDBS. The controller would have a scheduler that would choose the next request for execution. This request would then be broadcasted to all the back-ends for execution. We discard this approach from consideration because it aggravates the controller limitation problem which was singled out in Chapter 2 for the multiple back-end systems.

Another alternative would be to incorporate a centralized concurrency control mechanism at each back-end. Such an approach alleviates the controller limitation problem. However, we are questioning whether this approach is sufficient for a partitioned database with multiple back-ends such as MDBS.

Let us recall that in MDBS, a request is broadcasted to all the back-ends. Also, let us recall that MDBS executes requests in an MIMD fashion. That is, a request is not executed at exactly the same instant in all the back-ends. Rather, the request is being executed asynchronously in the different back-ends. With these observations, let us consider the following two requests that are being broadcasted by MDBS to each back-end.

U1: UPDATE (File = EMPLOYEE) <Salary = Salary + 2>

U2: UPDATE (File = EMPLOYEE) ∧ (Dept = 6) <Salary = Salary * 2>

We also consider for our example that an update U0 has been broadcasted by MDBS to the back-ends prior to either U1 or U2. Let U0 be

U0: UPDATE (File = EMPLOYEE) $\wedge$ (Dept = 7) <Salary = Salary + 2>

We see that U0 and U1 are incompatible because they may simultaneously try to update the records belonging to department 7. However, updates U0 and U2 are compatible, since they update different sets of records. Now, let there be two back-ends in MDBS. When updates U1 and U2 are received at back-end 1, let us assume that U0 has not been completed at that back-end. Hence, back-end 1 cannot execute U1, since U0 and U1 are incompatible. However, it may start executing U2, since U0 and U2 are compatible. Back-end 1 may execute U1 at some later time, after U0 and U2 have been completed. Hence, the order of execution of updates U1 and U2 at back-end 1 is U2 followed by U1. On the other hand, let us assume that U0 has completed at back-end 2 when U1 and U2 are received. This is possible because U0 is not executed at exactly the same time instant in both the back-ends. Furthermore, U0 may incur less processing at back-end 2 due to fewer employee records stored at the back-end. Thus, back-end 2 may execute U1. It will then execute U2 at some later time. Thus, the order of execution of updates at back-end 2 is U1 followed by U2. In spite of the fact that inter-consistency is maintained at each back-end, the result at back-end 1 is computed in terms of ((value * 2) + 2) and the result at back-end 2 is computed in terms of ((value + 2) * 2). These results are likely to be different. Consequently, the state of the MDBS database is inconsistent.

From this illustration, it is obvious that the guarantee of inter-consistency at each back-end is not sufficient to guarantee the consistency of a partitioned database utilizing multiple back-ends.

6.3 Monolithic Consistency and Non-Permutable Requests

From the previous example, we make three more observations. We first observe that the inconsistent state of the MDBS database would not occur if the entire database were stored at a single back-end and the requests issued to MDBS were executed by the single back-end in some serial order. We also observe that the inconsistency occurred because the two back-ends executed two requests in different orders and because different execution orders for these two requests lead to different results at these back-ends. Thus, the inconsistency may be avoided by the following techniques. Whenever two requests r1, r2 are such that the result of executing r1 followed by r2 is different from the result of executing r2 followed by r1, we term them non-permutable requests and ensure that these two requests are executed in the same order at all back-ends. Thus, to maintain database consistency for multiple back-ends, we say that we must maintain the same execution order among all non-permutable requests at all back-ends. Finally, we observe that inter-consistency must also be maintained at all back-ends. Otherwise, a back-end will try to simultaneously execute incompatible requests and this leads to problems of lost updates, etc. [Gard77].

In summary, we need to preserve partitioned database consistency for multiple back-end systems such as MDBS. This is termed monolithic consistency. In order to preserve monolithic consistency, we need to maintain an execution order among non-permutable requests and to preserve inter-consistency at all back-ends. To maintain an order among non-permutable requests, we need to identify pairs of requests that are non-permutable. To preserve inter-consistency, we need to identify pairs of requests that are incompatible. The identification of request pairs that are non-permutable and incompatible constitutes our major contributions in the following sections. First, however, let us develop some notations and terminology.

6.4 Notations and Terminology

In order to simplify the ensuing discussion, we use the following notation. Insert requests are of the form

INSERT R

where R is some record to be inserted. Delete requests are of the form

DELETE Q

where Q is the query used to select the records to be deleted. Retrieve requests are of the form

RETRIEVE Q <A>

where Q is the query used to select the records to be retrieved and <A> is the target-list of attributes whose attribute values will be fetched from the records retrieved. The By-Clause and the WITH-Clause are removed from the representation of the retrieve request because their presence is immaterial to this discussion on concurrency control. Update requests are one of the three forms

UPDATE Q A,1

UPDATE Q A,2

UPDATE Q A,3

[illegible] update request represents an update with a type-0 [illegible] form represents an update with a type-I modifier and [illegible] represents an update with a type-II modifier. We recall, from [illegible] update request with a modifier of type-III or IV is actu- [illegible] MDBS as a retrieve request followed by an update request with [illegible] type-0. Hence, we do not need to consider update requests with modifier of type-III or IV. In the three forms of the update request shown above, Q is the query used to select the records to be updated and A is the attribute being modified. If we do not wish to distinguish between the three forms of the update, we will write

UPDATE Q A

We will say that two queries Q1 and Q2 are disjoint if the records which satisfy Q1 and the records which satisfy Q2 do not have a record in common. For example, (Salary > 50) and (Salary < 50) are disjoint. If two queries are not disjoint, they are said to overlap. We will also say that attribute A (being modified) belongs to query Q, if A is also one of the attributes in the predicates of Q. Finally, we say that attribute A belongs to target-list <Y> if A is also one of the attributes in <Y>.

6.5 Request Permutabilities

Using the notation and terminology of the previous section, we will now present pairs of requests which are (or could be) permutable. As we recall, a pair of requests r1, r2 is (could be) non-permutable if the result of executing r1 followed by r2 is (could be) different from the result of executing r2 followed by r1. We present, below, ten pairs of such requests. Of these, pairs (4), (7) and (10) represent pairs of requests that could be non-permutable and the remaining ones are pairs of requests that are non-permutable.

(1) INSERT R
    RETRIEVE Q <A>, where R satisfies Q.

An example of the pair is

```
INSERT (<Salary,1000>, <Dept=TOY>)
RETRIEVE (Salary>100) <Dept>
```

(2) DELETE Q1

RETRIEVE Q2 <A>, where Q1 and Q2 overlap.

An example of the pair is

DELETE (Salary>50)

RETRIEVE (Salary>25) <Dept>

(3) UPDATE Q1 A

RETRIEVE Q2 <A>, where Q1 and Q2 overlap and A belongs to <A> but not to Q2.

For example,

UPDATE (Salary>50) <Rank=25>

RETRIEVE (Salary>50) <Rank>

(4) UPDATE Q1 A

RETRIEVE Q2 <A1>, where A belongs to Q2.

For example,

UPDATE (SALARY>25) <Salary=Salary-2>

RETRIEVE (Salary<25) <Rank>

(5) INSERT R

DELETE Q, where R satisfies Q.

For example,

INSERT (<Salary,1000>, <Dept,TOY>)

DELETE (Salary>50)

(6) INSERT R

UPDATE Q A, where R satisfies Q.

For example,

INSERT (<SALARY,1000>, <Dept,TOY>)

UPDATE (Salary>50) <Dept=SALES>

(7) UPDATE Q1 A

DELETE Q2, where A belongs to Q2.

For example,

UPDATE (Salary>25) <Salary=Salary-2>

DELETE (Salary<25)

(8) UPDATE Q1 A,2

UPDATE Q2 A,2, where Q1 and Q2 overlap and the type-I modifier of one update consists of an addition or subtraction and the type-I modifier of the other update consists of some multiplication or division.

For example,
UPDATE (Salary>25) <Rank=Rank + 1>
UPDATE (Salary>50) <Rank=Rank * 2>

(9) UPDATE Q1 A,1
UPDATE Q2 A,2, where Q1 and Q2 overlap, and modifiers are of types 0 and I, respectively.
For example,
UPDATE (Salary>25) <Rank=25>
UPDATE (Salary>50) <Rank=Rank + 1>

(10) UPDATE Q1 A
UPDATE Q2 A,2, where A belongs to Q2.
For example,
UPDATE (Salary>50) <Salary=Salary + 25>
UPDATE (Salary>75) <Rank=Rank + 1>

Note that there are two pairs of non-permutable requests in which one request is an update and the other request is a retrieve. These are the pairs (3) and (4). In pair (3), the retrieve request will retrieve the same set of records whether it is executed before or after the update. However, if it is executed after the update, the modifications made by the update will be visible in the retrieved records and these modifications will not be there if the retrieve request is executed before the update. In pair (4), the retrieve request will actually retrieve a different set of records if it is executed before the update than it will if it is executed after the update.

As we have said before, our concurrency control algorithm must maintain an ordering among all non-permutable requests. However, a little thought shows us that this does not apply to all cases. For example, record insertion takes place only at a single back-end. An ordering needs to be maintained only among non-permutable requests which are executed at all back-ends. Thus, an ordering needs to be maintained only among all non-permutable requests which are not inserts. To put it another way, our concurrency control algorithm will not need to check for pairs (1), (5) and (6) of non-permutable requests.

Next, let us consider the pairs of incompatible requests.

6.6 Request Compatibilities

To repeat, two requests r1 and r2 are compatible if their simultaneous execution gives the same result as would be obtained if we execute in the sequence r1 followed by r2 or in the sequence r2 followed by r1. Clearly, for two requests to be compatible, they must also be permutable. However, two requests may be incompatible without being nonpermutable. For instance,

UPDATE (Salary>50) <Salary=Salary + 2>
UPDATE (Salary>50) <Salary=Salary + 2>

are incompatible but permutable. The two requests are incompatible because their simulatenous execution can lead to the problem of lost updates. However, the requests are permutable because the result of executing these updates in any order is the same.

For MDBS, it may be easily verified that the pairs (1) through (7) of non-permutable requests are also pairs of incompatible requests. Furthermore, the three pairs of requests, (8) to (10), are also incompatible. They are repeated here with new examples.

(8) UPDATE Q1 A
UPDATE Q2 B, where Q1 and Q2 overlap.
For example,
UPDATE (Salary>25) <Rank=Rank + 1>
UPDATE (Salary>50) <Rank=Rank + 1>

(9) UPDATE Q1 A
UPDATE Q2 B, where A belongs to Q2
For example,
UPDATE (Salary>25) <Rank=Rank + 1>
UPDATE (Rank=20) <Department=17>

(10) UPDATE Q1 A
DELETE Q2, where Q1 and Q2 overlap.
For example,
UPDATE (Salary>25) <Rank=Rank + 1>
DELETE (Salary>50)

6.7 Cluster-Based Permutabilities and Compatibilities

In the previous sections, we described how to identify pairs of non-permutable and incompatible requests. Such identification requires, among other things, a procedure for testing if two queries overlap. Such a procedure can be quite complex. Hence, the procedures for identifying pairs of non-permutable and incompatible requests may be quite complex. In this section, we shall use the notion of clusters to simplify the procedures for identifying non-permutable and incompatible requests.

We recall, from Chapter 5, that each request in a transaction is first attached with an authorized cluster (for an insert request) or with a set of authorized clusters (for a non-insert request). In the ensuing discussion, we will represent the attached requests as follows. An attached insert request is represented by

INSERT P

where P is the authorized cluster for the request. An attached retrieve request is represented by

RETRIEVE <P>

where <P> is the set of authorized clusters for the retrieve request. Similarly, an attached delete request is represented by

DELETE <P>

In the case of an update request, we term the <u>set of future clusters</u> as the set of all those clusters into which records may be placed after the update in accordance with the request. Then, an update request is represented by

UPDATE <P>

where <P> is the set of clusters formed as the union of the set of authorized clusters and the set of future clusters for the update request.

Now, pair (1) of non-permutable requests of Section 6.5 becomes

(a) INSERT P
RETRIEVE <X>, where P belongs to <X>.

Pair (2) of the non-permutable requests of Section 6.5 becomes

(b) DELETE <X>
RETRIEVE <Y>, where <X> and <Y> have non-null intersection.

Pairs (3) and (4) of the non-permutable requests of Section 6.5 becomes

(c) UPDATE <X>
RETRIEVE <Y>, where <X> and <Y> have non-null intersection.

Pair (5) of the non-permutable requests of Section 6.5 becomes

(d) INSERT P
RETRIEVE <X>, where P belongs to <X>.

Pair (6) of the non-permutable requests of Section 6.5 becomes

(e) INSERT P
UPDATE <X>, where P belongs to <X>.

Pair (7) of the non-permutable requests of Section 6.5 becomes

(f) UPDATE <X>
DELETE <Y>, where <X> and <Y> have non-null intersection.

Finally, pairs (8), (9) and (10) of the non-permutable requests of Section 6.5 become

(g) UPDATE <X>
UPDATE <Y>, where <X> and <Y> have non-null intersection.

Next, let us consider the ten pairs of incompatible requests of Section 6.6. The first seven pairs of Section 6.6 are identical to those of Section 6.5 and are therefore translated into (a) through (f) as above. Pairs (8) and (9) of Section 6.6 become pair (g) above. Finally, pair (10) of Section 6.6 becomes pair (f) above.

Thus, we now have a simpler decision procedure for identifying pairs of incompatible and non-permutable requests. This procedure requires us to first determine for a request the authorized cluster (for an insert request) or the set of authorized clusters (for delete and retrieve requests) or the sets of authorized and future clusters (for an update request). The algorithm for determining the authorized cluster (for an insert request) or the authorized set of clusters for(delete, retrieve and update requests) has been described in Chapter 5. We shall now describe the algorithm for determining the set of future clusters for an update request of the form

UPDATE Q A

This algorithm will utilize the set of authorized clusters already calculated.

6.7.1 Determining the Set of Future Clusters for An Update Request

Consider a record in one of the authorized clusters. We are trying to determine the cluster to which the record will belong after being updated by the request, given that we know the cluster to which that record belongs before the update. In other words, we know the set of descriptors which defines the (authorized) cluster of a record before it is updated and we also know the update request in consideration. From these two pieces of information, we are trying to determine the set of descriptors which will define the (future) cluster of the record after the record is updated.

A. Determining the Future Cluster of a Record to be Updated Without Having Seen the Record

First, we note that updates in MDBS modify only a single attribute-value. Second, we recall that the cluster to which a record belongs is defined by the set of descriptors from which every directory keyword (attribute-value) in the record is derivable. From the above observations, we conclude that the set of descriptors which defines the cluster of a record before it is updated and the set of descriptors which will define the cluster of the record after it is updated can differ in at most one descriptor. This is the descriptor from which the keyword in the record containing the attribute being modified is derivable. Therefore, we may do the following to determine the set of descriptors which will define the (future) cluster to which the updated record will belong without first examining the record. The motivation in pursuing the determination of the future cluster for a record (prior to the examination of the record) is to lock up this future cluster for subsequent retrieval and update of the record.

Consider the set of descriptors which defines the cluster to which the record belongs before update. Delete, from this set of descriptors, the descriptor from which the keyword containing the attribute being modified is derivable. In this case, the keyword is of course a directory keyword. (See Section 4.1.1.) If the attribute being modified is not contained in any directory keyword, then there is no descriptor to be deleted. In either case, we must add a new descriptor to the set. This new descriptor is the one from which the keyword containing the attribute being modified will be derivable. If the modifier in the update request is of type-0, this new descriptor to be added may be easily determined, since the new value to be taken by the attribute being modified is specified in the modifier of the update request itself. However, if the modifier in the update request is not of type-0, the new value to be taken by the attribute being modified cannot be determined by examining the update request itself. Therefore, we cannot add the new descriptor to the set of descriptors. In this case, we only have a partial set of descriptors which may define several clusters each of which could potentially be used to contain the newly updated record. We will determine every such potential (future) cluster by searching the augmented CDT for clusters whose descriptor sets properly contain the partial set of descriptors at hand.

B. Determining All the Future Clusters in Order to Lock Them Up For Record Updates

Since there are many records to be updated and these records are in those authorized clusters already determined, we will consider the other records to be updated with the above procedure. In other words, by repeating the above procedure for records in all the authorized clusters, we may determine the set of potential future clusters. For every cluster k in the set of potential future clusters so determined, MDBS accesses the entry in the augmented cluster definition table (CDT) corresponding to the row for cluster k and the column for the user id of the user that issued the update request. If this entry contains a field-level access control (see Chapter 5) with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, MDBS will remove cluster k from the set. The above procedure is repeated for every cluster in the set. The remaining set of clusters is the set of future clusters.

More formally, let the update be of the form

UPDATE Q A,

be issued by user U. If A, the attribute being modified, is not a directory attribute, the set of future clusters is identical to the set of authorized clusters. Otherwise, the set of future clusters {F} will have to be derived from the set of authorized clusters {P}. Let the members of {P} be P1, P2,...,Pk and let their corresponding descriptor sets be D1, D2,...,Dk.

If the modifier in the update request is of type-0, extract the constant V specified in the modifier, form the attribute-value pair <A,V> and locate the descriptor D corresponding to <A,V> from the augmented DDIT. For each descriptor with A as its attribute, replace that descriptor in Di with D. If Di does not contain a descriptor with A as its attribute, add the descriptor set DDi for Di. Now, use the descriptor set DDi to search the augmented cluster definition table (CDT) for clusters whose definitions include DDi. Place all such clusters into the set of future clusters {F}. For each cluster k in {F}, access the entry in the augmented CDT corresponding to cluster k and user U. If the entry contains a field-level access control with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, remove cluster k from {F}. By repeating the above procedure for every Di, the set of future clusters {F} is completely determined.

Now let us describe how {F} may be determined when the modifier in the update request is of type-I or type-II. We do the following for each descriptor set Di, i from 1 to k. If Di contains a descriptor with A as its attribute, remove that descriptor from the descriptor set Di. If Di does not contain a descriptor with A as its attribute, then do nothing to Di. In either case, form the descriptor set DDi from Di. Now, use the descriptor set DDi to search the augmented CDT for clusters whose definitions include DDi. Place all such clusters into the set of future clusters {F}. For each cluster k in {F}, access the entry in the augmented CDT corresponding to cluster k and user U. If the entry contains a field-level access control with an unspecified attribute combination part and with 'No-Insert' in the disallowed access part, remove cluster k from {F}. By repeating the above procedure for every Di, the set of future clusters {F} is completely determined.

Finally, if the update request has a modifier of type-III or type-IV, it is executed as a retrieved request followed by an update request with a modifier of type-O.

C. Determining Incompatible and Non-Permutable Requests

Once we have determined the sets of authorized and future clusters, the pairs (a) to (g) of incompatible and non-permutable requests may be easily determined.

6.7.2 A Case of 'Over-Determination'

The above simple procedure for identifying pairs of incompatible and non-permutable requests has been achieved at a price. This is because our procedure may sometimes identify a pair of requests as non-permutable (incompatible) even if it is actually permutable (compatible). This is because our procedure for determining the set of future clusters actually produces every potential cluster to which a record may belong after update. However, in reality the updated records can not belong to every potential cluster so determined but to a single cluster. In a pair of requests - see (g) - the possibility of having non-null intersection with a larger number of potential clusters is higher than with a single cluster. Consequently, the pair may be considered non-permutable (incompatible) due to some non-null intersection over potential clusters of the paired requests. However, if the non-null intersection does not contain the single cluster to which the record eventually belongs, then the pair is neither non-permutable nor incompatible. Our price is therefore in over-classification of the requests. We believe, nevertheless, that this price is worth paying because of the resulting simplification in the procedure for identifying pairs of incompatible and non-permutable requests.

6.8 The Cluster-Based Concurrency Control Algorithm

In the previous section, we described the procedures for identifying pairs of incompatible and non-permutable requests. In this section, we will describe the entire concurrency control algorithm which uses these procedures.

As described in Chapter 5, all the requests in a transaction are partially executed until the set of authorized clusters for each request in the transaction has been determined. In the case of update requests, they are partially executed until the set of authorized clusters and the set of future clusters have both been determined. Each request in a transaction may then be attached with its authorized cluster, the set of authorized clusters or the set of authorized and future clusters. For simplicity, we will refer to these cluster(s) that attached to a request as *the cluster(s) needed by the request*. As described in Chapter 5, these attached requests are used to form a secure transaction. In the sequel, we refer to the union of all the clusters needed by the attached requests of a secure transaction as the *clusters needed by a secure transaction*. Further, we refer to a secure transaction as a transaction and to an attached request as a request.

We will see that a transaction will need to lock various clusters during the course of its execution in one of four *modes* and in one of two *categories*. The four modes in which a transaction may lock a cluster are retrieve, delete, insert and update. The two categories in which a transaction may lock a cluster are the to-be-used category and the being-used category. When a transaction wishes to lock a cluster, it must specify both the mode and the category in which it wishes to lock the cluster.

6.8.1 Incompatible and Non-permutable Locks

Consider a request r1 in transaction t1 and a request r2 in transaction t2. Let c be one of the clusters needed by r1 and also one of the clusters needed by r2. Then, both t1 and t2 will need to request locks on cluster c. The lock requested by transaction t1 on cluster c is said to be incompatible (non-permutable) with the lock requested by transaction t2 on the same cluster, if request r1 of t1 and request r2 of t2 form a pair of incompatible (non-permutable) requests. By considering the pairs of incompatible (non-permutable) requests of Section 6.7, we may easily derive the pairs of incompatible (non-permutable) locks shown in Figure 42. For instance, Figure 42 shows that in any category a lock on cluster i in the delete mode is incompatible and permutable with a lock on cluster i in the insert mode for all i. Now, our concurrency control algorithm, instead of using the pairs of requests of Section 6.7, may use the lock table of Figure 42 to determine incompatible (non-permutable) locks and hence, incompatible (non-permutable) requests. From Figure 42, we see that only the modes of two locks (and not their categories) are used in deciding if they are incompatible and non-permutable. Hence, in the sequel, we can also talk of incompatible and non-permutable lock modes. We are now ready to describe the execution sequence of a transaction.

6.8.2 The Execution Sequence of a Secure Transaction

The execution sequence of a transaction begins when the transaction locks all the clusters needed by the transaction in the to-be-used category. The mode in which a cluster is locked depends on the type of request to be executed on that cluster. Thus, if cluster i is needed by a retrieve (delete, insert or update) request, it will be locked in retrieve (delete, insert or update) mode. If a cluster i is needed by more than one request in the transaction, it may be locked in more than one mode. The locking of clusters by a transaction is performed independent of whether or not other transactions have locked the same clusters in any mode or category.

Now, execution of the requests of the transaction begins. Before a retrieve (delete, insert or update) request in a transaction may be executed, the clusters

	Delete Lock	Insert Lock	Update Lock	Retrieve Lock
Delete Lock	C,P	I,P	I,N	I,N
Insert Lock	I,P	C,P	I,P	I,P
Update Lock	I,N	I,P	I,N	I,N
Retrieve Lock	I,N	I,P	I,N	C,P

I:	Incompatible
C:	Compatible
N:	Non-permutable
P:	Permutable

Figure 42. The Lock Table

needed by the request must be locked in retrieve (delete, insert or update) mode and in the being-used category. Since the clusters have been locked in some mode in the to-be-used category, the appropriate locks must be converted from the to-be-used category to being-used category. A lock on cluster i may be converted only if no other incompatible lock is held on cluster i by another transaction in the being-used category and no other non-permutable lock is held on cluster i by an earlier transaction in any category. An earlier transaction is one which has arrived earlier than the arrival of the present transaction at the back-end. The first check ensures inter-consistency and the second check ensures an ordering among non-permutable requests. Together, the two checks ensure monolithic consistency. After the appropriate locks have been converted, the request may be executed. Then, the locks needed by the request are released. The above procedure is repeated for every request in the transaction until all the requests in the transaction are executed.

6.8.3 The Concurrency Control Mechanism

In the actual implementation, each back-end will maintain one queue for each cluster stored at that back-end. Each element in a cluster queue (say, for cluster k) will contain three pieces of information. First, it will contain a transaction number. Second, it contains the mode in which the transaction is holding (or wishes to hold) a lock on cluster k. Third, it contains the category in which the transaction is holding (or wishes to hold) a lock on cluster k.

These cluster queues will be consulted by MDBS whenever locks are requested by transactions. For instance, when a transaction locks all clusters in the to-be-used category, MDBS will create an element for each cluster needed by it and the above elements will be placed in the appropriate cluster queues. Similarly, whenever a lock conversion is requested by a transaction, MDBS consults these cluster queues. Thus, when transaction i requests the conversion of a lock on cluster k, the system will search the queue for cluster k. If an element for a transaction holding an incompatible lock in the being-used category is found, the conversion cannot be granted. Likewise, if an element for an earlier transaction holding a non-permutable lock in any category is found, the conversion cannot be granted. Whether the conversion is granted or not, the category of the queue element for the transaction is changed from to-be-used to being-used. However, if the conversion is not granted, the transaction is deactivated. That is, its execution is suspended. The deactivated transaction will later be reactivated when the transaction holding the incompatible or non-permutable lock releases that lock. A reactivated transaction will continue execution from the point where it was suspended.

The cluster queues are also consulted whenever a transaction releases a lock on a cluster. In this case, the appropriate element will be removed from the queue for that cluster. Furthermore, the system will search the queue for that cluster to determine if any of the locks requested by other transactions may now be granted. If so, the transactions whose locks may now be granted are activated.

Three algorithms are presented here. The first algorithm is executed when a transaction is first received at a back-end and requests to lock all clusters needed by the transaction in the to-be-used category. The second algorithm is executed whenever a transaction wants to convert a lock request from the to-be-used category to the being-used category. The third and final algorithm is executed whenever a transaction releases all the locks needed by one of the requests in the transaction. We use the following notation to simplify the statement of the algorithms.

Q(i)	:	The queue of transactions (more precisely, transaction numbers of the transactions) on cluster i.
Q(i,j)	:	The j-th element in Q(i).
MODE(i,k)	:	The mode in which the k-th element in Q(i) is holding a lock.
CAT(i,j)	:	The category of the lock specified in element Q(i,j).
TRAN(i,j)	:	The transaction number in Q(i,j).

Let COM(A,B) imply that lock modes A and B are compatible and let PER(A,B) imply that the lock modes are permutable. Similarly, let NOTCOM(A,B) imply that lock modes A and B are incompatible and let NOTPER(A,B) imply that lock modes A and B are non-permutable. The j-th element is removed from Q(i) by "Remove Q(i,j)" and placed as the last element of the queue by "Into Q(i)". We now present these three algorithms, known as Algorithm Initialize, Algorithm Lock-Convert and Algorithm Lock-Release in Figure 43.

In the above discussion, we assume that a separate queue would be maintained for each cluster stored at a back-end. An alternative would be to consolidate all these queues into a single queue at each back-end. An element in such a consolidated queue represents a lock that a transaction holds (or wishes to hold) and must contain four pieces of information. The four pieces of information

```
ALGORITHM INITIALIZE
   /* Executed when a transaction is first received at a back-end*/
   /* C(i) denotes the set of clusters needed for request i in transaction T,*/
   T = Transaction Number
   Num = Number of Requests in Transaction.
   Create a queue element QE with CAT = 'to be used' and TRAN = T.
   For i=1 step 1 until NUM do
      begin
         Calculate the set C(i)
         Let M = mode in which the clusters in C(i) are locked
         Augment QE with MODE = M.
         For each element j in C(i) Insert QE into Q(j)
      end
END ALGORITHM INITIALIZE


ALGORITHM LOCK CONVERT
   /* Executed when a transaction T wants to convert locks for request R in
      mode M from 'to be used' to 'being-used'*/
   Calculate the set C(R) /* The set of clusters needed by R*/
   For each element j of C(R)
      begin
         let T be Q(j,k)
         for i=1 to k-1 step 1
            begin
               If NOTCOM(MODE(j,i),M) and CAT(j,i) = 'being-used' then
                                                    deactivate T
               If NOTPER(MODE(j,i),M) and CAT(j,i) = 'to-be-used' then
                                                    deactivate T
            end
         CAT(j,k) = 'being-used'
      end
END LOCK CONVERT


ALGORITHM LOCK RELEASE
   /* Executed when a transaction T wants to release locks for request R in
      mode M*/
   Let C(R) be the set of clusters needed by R.
   For each element j of C(R)
      begin
         let there be k elements in Q(j)
         MODE=MODE(j,1)
         CHECK=TRUE
         For i=1 to k while CHECK
            begin
               If COM(MODE(j,1),MODE) then
                                       begin
                                          activate TRAN(j,1)
                                          remove Q(j,1)
                                       end
                                 else CHECK=FALSE
            end
      end
END LOCK RELEASE
```

Figure 43. Three Algorithms for Cluster Queue Management

needed in a consolidated queue element are the transaction number of the transaction holding (or wishing to hold) the lock, the cluster number on which the transaction holds (or wishes to hold) the lock, and the mode and category of the lock. The advantage of such a scheme over the previous scheme where several queues were maintained is that, now, each back-end needs to maintain only a single queue. The disadvantage of such a scheme over the previous one is that in order to look for incompatible and non-permutable locks, every element in this consolidated queue must be searched. Compare this to the previous scheme where only the queue of elements for a specific cluster has to be searched. The disadvantage arises because the number of elements in the queue for a specific cluster is likely to be much less than the number of elements in the consolidated queue. However, the consolidated queue scheme may be preferable when the number of transactions expected in MDBS at any instant of time is not very large. In this case, the consolidated queue is not expected to be large and we would prefer to maintain a single consolidated queue rather than several cluster queues most of which will probably be empty.

The three algorithms presented in this section, together with the execution procedure presented in the previous section constitute the logic of the MDBS concurrency control mechanism which is distributed among the back-ends. The queues and the lock tables are the data structures of the mechanism.

6.9 An Examination of the Concurrency Control Mechanism

The MDBS concurrency control mechanism is applicable to both partitioned and centralized databases. It also differs from other concurrency control mechanisms in the way that locks are utilized. The MDBS mechanism algorithm uses four lock modes rather than the traditional two lock modes. By separating insert and delete locks from update locks, the MDBS concurrency control mechanism is able to support a greater degree of concurrency than would otherwise be possible.

6.9.1 New Solutions for Centralized-Database Concurrency Control

At a theoretical level, one of the contributions of this work has been in the identification of permutable and compatible requests, where the requests are issued in a high-level, query-based language. The authors of [Gard77] also identify permutable and compatible requests, but they do so for a language which is not query-based. On the other hand, the authors of [Eswa76] do consider a query-based language. However, they only identify compatible requests and they do not identify permutable requests for their query-based language. Furthermore, they do not differentiate insert and delete requests from update requests.

At a practical level, a method for enforcing locking when a query-based language is employed has been presented. The scheme is simpler than predicate locking [Eswa76] and is based on cluster locking. Two factors have contributed to the simplicity of our locking procedure over the procedure suggested in [Eswa76]. First, in [Eswa76] a query which requires locking must be checked against all existing queries. This can be time-consuming. On the other hand, MDBS mechanism does not require checking against all the outstanding queries. This is because a query is converted into a corresponding set of clusters and checking of the locks on these clusters is then performed. Secondly, the procedure in [Eswa76] for checking if two queries conflict requires converting the queries to disjunctive normal form. The time for such conversion can be exponential for some types of queries [Sava76]. In MDBS, the procedure for converting queries into a corresponding set of clusters however does not require conversion of the queries into disjunctive normal form. The procedure for determining the set of corresponding clusters of a query which is not in disjunctive normal form is presented in Appendix G. It is clear from this procedure that the query does not need to be converted into disjunctive normal form. Therefore, the procedure is not exponential in the size of the query and, in fact, is linear in the size of the query. For these two reasons, we believe that the MDBS concurrency control mechanism is superior to that of [Eswa76].

We note also that our concurrency control algorithm is deadlock-free. This is because a transaction can never obtain a lock unless it is confirmed that all earlier transactions will never need that lock or will need it only in a permutable mode. Hence, a transaction can only be blocked by earlier transactions and can never be blocked by subsequent transactions, thus avoiding deadlock. This is unlike the method of [Eswa76] or that of [Gard77], both of which may lead to deadlocks.

As a result, one of the disadvantages of the solutions of [Eswa76] over ours is that their solutions do not adequately cover the issue of starvation. That is, since transactions have to be backed-up and restarted in the event of deadlock, there must be enough evidence (say, a formal proof) to show that they will eventually complete. Such evidence is not provided in [Eswa76] or [Gard77]. Our system cannot suffer from starvation, since it is deadlock-free and does not require transaction restart.

If all the requests in transactions of MDBS are retrieve requests, the transactions will run without interfering with each other. Similarly, if all the requests in transactions are inserts, or all the requests in transactions are deletes, the transactions will run without interfering with each other. In this respect, our concurrency control mechanism is better than all others which do not make this distinction between inserts, deletes and updates.

Finally, unlike the method presented in [Eswa76], the mechanism presented here does not follow the two-phase protocol [Eswa76]. A two-phase protocol is one in which a transaction must not acquire new locks after it has given up any lock. That is, it must acquire all the locks it needs in the beginning and then release these locks one by one, or it must acquire the locks one by one as it needs them and then release them altogether in the end. A mechanism which uses the two-phase protocol will achieve a lower degree of concurrency than a mechanism such as ours where the locks are acquired and released as needed and where the protocol is not two-phase.

6.9.2 New Solutions for Partitioned-Database Concurrency Control

As far as we are aware, there are only two other published solutions to the problems of concurrency control for partitioned databases. These are the solutions of [Dewi80] and that of System D which is also reported in [Dewi80].

Before we can compare our solution to these two solutions, we need to develop some terminology. In these two solutions, each transaction has its own private buffer in which it does updates. The updates made by a transaction t cannot be seen by other transactions until t writes the contents of its private buffer onto the secondary memory. At this time, t is said to commit and its updates are said to become visible. Furthermore, it will be seen that these two solutions require transactions to read, update and write entities, where the entities are attribute-values. The reading, updating and writing of entities are referred to as actions of a transaction. A transaction t1 is said to come before a transaction t2 in the serialization order if

(a) t1 reads an entity which t2 writes later, or

(b) t1 writes an entity which t2 reads later, or

(c) t1 writes an entity which t2 writes later.

It is clear from the definition that the fact that t1 comes before t2 does not necessarily imply that t2 comes before t1. Finally, a conflict is said to occur between t3 and t4 if t3 comes before t4 and t4 comes before t3.

Let us briefly try to describe the solutions of [Dewi80] and System D. These solutions do not employ locking for concurrency control. They wait until a transaction is ready to commit. When a transaction is ready to commit, the system will try to detect any conflicts. If no conflicts are detected, the transaction is allowed to commit. Otherwise, a number of alternative actions may have to be taken by the system. Three alternative actions are proposed in [Dewi80]. The first leads to the so-called _starvation solution_, the second leads to the _non-starvation solution_ and the third leads to the _restrictions-list solution_. The starvation solution may cause transactions to wait perpetually as pointed out in [Dewi80]. Furthermore, we will show, by means of an example, that the starvation solution is erroneous in that it may result in an inconsistent database state. The non-starvation solution, on the other hand, causes transactions to be backed-up unnecessarily. Therefore, the restrictions-list solution of [Dewi80] is the best of the three solutions. Briefly, the restrictions-list solution is as follows. When a transaction commits, a list of prohibited actions will be immediately formed and associated with all other active transactions in the system. These prohibited actions, which are actions that will cause a conflict, are stored in a restrictions list, one list per active transaction. When a transaction tries to perform an action in its restrictions list, the transaction is backed-up and restarted. This completes our explanation of the restriction-list solution of [Dewi80]. The solution of System D may be considered as a distributed version of the restrictions-list solution of [Dewi80]. Having briefly explained these two solutions, we are now ready to compare the MDBS solution with these two solutions.

A. Rich Semantics in DML and New Concept of Permutability

The first weakness of these two solutions is that they use a simple data manipulation language which allows only for reading and writing entities. MDBS on the other hand, employs a query-based data manipulation language (DML). Secondly, these systems make no effort to identify permutable requests. Because of these two weaknesses, these two systems will cause transactions to be backed-up unnecessarily. Consider the following scenario. Let there be two transactions t1 and t2. t1 increments the value of x by 2 and the value of y by 3. t2 increments the value of x by 3. That is, t1 consists of the following

```
t1:  read x
     x = x + 2
     write x
     read y
     y = y + 3
     write y
     commit
```

Also, t2 consists of

```
t2:  read x
     x = x + 3
     write x
     commit
```

Furthermore, let us assume that the actions of t1 and t2 are interspersed in the following manner.

```
t1:  read x
t2:  read y
t1:  x = x + 2
t1:  write x
t1:  read y
t1:  y = y + 3
t1:  write y
t1:  commit
t2:  read x
t2:  x = x + 3
t2:  write x
t2:  commit
```

Both the restrictions-list and System D solutions would cause t2 to be backed-up after t1 commits. That is, the actions after the action, t1: commit, shown above will not be allowed by these two solutions. This is because there is a conflict between t1 and t2. Let us elaborate on the conflict. In the serialization, t2 comes before t1. This is due to the fact that t2 has read entity y before the update of y by t1 became visible. However, t1 also comes before t2, when t2 tries to read x after t1 has written x. Hence, there is indeed the conflict. Consequently, t2 will be backed up.

With our understanding of the concept of permutability among requests, we observe that since t1 was reading x only to update it and since the two updates on x are permutable, there is no need to back-up t2. This is because the result of the execution sequence shown above is exactly the same as the result that would have been produced if t2 came before t1 in the serialization. The unnecessary back-up is caused because the two proposed solutions of [Dewi80] and System D do not differentiate between reading an entity for the purpose of transmitting the value to the user and reading an entity for the purpose of updating its value. This is due to a lack of semantics in the data manipulation language of System D. Besides the lack of semantic difference between these two kinds of reads, there is also the lack of any concept of permutable updates. For these two reasons, the proposed solutions of [Dewi80] and System D [Dewi80] failed to allow a perfectly 'legal' execution sequence to proceed.

B. Better Throughput and Lower Control Message Traffic

MDBS concurrency control mechanism allows transactions to commit at different times in different back-ends. Consequently, at a given instant, a transaction may have committed at one back-end and not committed at another back-end. This may happen if many more records have to be accessed at one back-end than at another one in order to satisfy the requests in the transaction. We believe that this leads to increased concurrency and therefore better throughput because each back-end is executing transactions at its own pace and does not have to wait for other back-ends to complete the execution of these transactions. In other words, the execution mode is truly MIMD. On the other hand, solutions such as those of [Dewi80] and System D, which require all back-ends to commit a transaction simultaneously are essentially SIMD.

To commit a transaction simultaneously, both the solution of [Dewi80] and that of System D require all the back-ends to exchange control messages. For example, in the case of [Dewi80], a special computer designated as the <u>concurrency</u> <u>control</u> <u>computer</u> will wait for n commit messages from the n back-ends before committing a transaction. Similarly, in the case of System D, each back-end must wait to receive (n-1) commit messages from the other (n-1) back-ends before committing a transaction. Control messages are also needed to broadcast the restrictions list to all the back-ends.

The MDBS solution, on the other hand, exchanges no messages among the back-ends for concurrency control. This is because neither are the transactions required to commit simultaneously nor is the restrictions list employed Hence, this solution serves to alleviate the control message traffic problem typified among the multiple back-end systems.

C. No Back-end Limitation Problem

Another problem with the solution of [Dewi80] but not with the solution of System D is that its concurrency control mechanism is implemented in a single dedicated back-end. This can lead to an unreliable system, should the back-end fail for whatever cause. It also violates our principle of distributing all the work among all the back-ends without specializing any back-end. The MDBS concurrency control mechanism does not have the back-end limitation problem.

D. A Question of Overhead Incurred During Concurrency Control

It is claimed in [Dewi80] that the aforementioned two solutions incur less overhead than locking-based solutions. However, this may be debatable. First of all, for each transaction, these solutions need to keep track of which entities have been read and written by what transactions. Is this any simpler than locking the entities that have been read and written by the transactions? Furthermore, these two solutions require the maintenance of restricitons lists which incur an overhead that cannot be found in MDBS.

E. Free From Starvation Errors

We now show that the starvation solution of [Dewi80] can lead to an inconsistent database. For instance, consider two transactions t5 and t6 issued to an employee database as follows.

```
t5:  read x
     y = 10
     write y
     x = x + 2
     write x
     commit
t6:  read y
     y = y + 2
     write y
     read x
     x = x * 3
     write x
```

That is, transaction t5 increments the value of x by 2 and changes the value of y to 10. Similarly, transaction t6 increments the value of y by 2 and multiplies the value of x by 3. The corresponding transactions T5 and T6 in MDBS might be

```
T5:  BOT
     UPDATE (File=EMPLOYEE) <y = 10>
     UPDATE (File=EMPLOYEE) <x = x + 2>
     EOT
T6:  BOT
     UPDATE (File=EMPLOYEE) <y = y + 2>
     UPDATE (File=EMPLOYEE) <x = x * 3>
     EOT
```

Before these two transactions are received the initial values of x and y in the database are assumed to be 5 and 6, respectively. Then, after the execution of these two transactions the final values of x and y will be 21 and 12, respectively, if t5 (T5) is executed before t6 (T6). Similarly, the final values of x and y will be 17 and 10, respectively, if t6 (T6) is executed before t5 (T5). Either of these two sets of final values for x and y leaves the database in a consistent state. Any other set of values for x and y leaves the database in an inconsistent state.

E.1 Transaction Execution by the MDBS Solution

Consider how these two transactions are executed in MDBS. Let us consider, for simplicity, that T5 was received before T6. Then, it is easy to see from the description of the MDBS concurrency control mechanism in Section 6.8 that MDBS will execute these two transactions in either one of the following sequences.

```
Sequence one
    T5:  UPDATE (File=EMPLOYEE) <y = 10>
    T5:  UPDATE (File=EMPLOYEE) <x = x + 2>
    T6:  UPDATE (File=EMPLOYEE) <y = y + 2>
    T6:  UPDATE (File=EMPLOYEE) <x = x * 3>
Sequence two
    T5:  UPDATE (File=EMPLOYEE) <y = 10>
    T6:  UPDATE (File=EMPLOYEE) <y = y + 2>
    T5:  UPDATE (File=EMPLOYEE) <x = x + 2>
    T6:  UPDATE (File=EMPLOYEE) <x = x * 3>
```

and will not permit any other execution sequence. Both the permitted execution sequences above leave the final value of x as 21 and the final value of y as 12. Therefore, the final state of the database is consistent irrespective of which execution seuqence is followed in MDBS.

E.2 Transaction Execution by the Starvation Solution

Now, let us consider how the transactions t5 and t6 are executed if the starvation solution of [Dewi80] is used for concurrency control. In that solution, the following execution sequence

```
t5:  read x
t6:  read y
t5:  y = 10
t5:  write y
t5:  x = x + 2
t6:  write x
t5:  commit
```

```
t6:  y = y + 2
t6:  write y
t6:  read x
t6:  x = x + 3
t6:  write x
t6:  commit
```

will be permitted except for the fact that t5 will not be allowed to commit until t6 commits. This is because t6 comes before t5 in the serialization order, since it reads an entity y which t5 will write later. After both t6 and t5 commit, the final values of x and y are 7 and 10, respectively. For the database to be consistent, the final values of x and y must be either 21 and 12 or 17 and 10. Hence, the database is in an inconsistent state. We conclude, therefore, that the starvation solution of [Dewi80] can lead to an inconsistent database.

6.10 The Execution of Incompletely-Specified Transactions

In this section, we will consider how the MDBS concurrency control mechanism may be extended to execute incompletely specified transactions. That is, we wish to execute a transaction even before all the requests in the transaction have been provided by the user. For instance, a transaction may consist of three requests. The user has specified only one request of the transaction. We want MDBS to start executing the one request without waiting for the user to provide the remaining two requests. First, we will show why our basic mechanism is not able to execute incompletely-specified transactions. Then, we will present two possible extensions to the basic concurrency control mechanism for executing incompletely-specified transactions.

Let transaction t1 begin with the following request

RETRIEVE (File=EMPLOYEE) (Salary)

Also let transaction t2 begin with the following request

UPDATE (File=EMPLOYEE) <Salary=Salary + 2>

Furthermore, let t1 be received before t2. Since t1 is received before t2, the retrieve request of t1 may be executed at each back-end. Now, the back-ends must decide whether or not to execute the update request of t2. The problem with executing the update request of t2 is as follows. After executing the update request of t2, MDBS may receive the second request of t1. For instance, it may be the following

UPDATE (File=EMPLOYEE) <Salary=Salary + 2>

This update request of t1 is non-permutable with the update request of t2. The basic concurrency control algorithm of MDBS requires us to ensure that whenever two non-permutable requests are received, the request of the earlier transaction is executed first and the request of the later transaction is executed next. By this principle, the update of t1 should have been executed before the update of t2 is executed because t1 is the earlier transaction. However, the update of t2 by now has already been executed, whereas the execution of the update of t1 has not yet begun. Therefore, we need to back-up transaction t2. Backing-up a transaction requires the transaction to give up all its locks and to start its execution all over again.

6.10.1 Problems With Backing Up Transactions

Consider the following record in our discussion

(<Employee,JAI>, <Salary,5000>).

Two transactions, t3 and t4, increment the salary of employee JAI by 100. Consider that t3 first increments the salary to 5100 and t4 increments the salary then to 5200. Assume, also that both t3 and t4 consist of other requests and that a back-up of t3 is required, after t3 and t4 both update the salary attribute of the record and before either t3 or t4 completes its transaction. In order to back-up t3, we need to undo all the changes made by t3, thereby restoring the database to the state prior to the execution of t3. In particular, we need to restore the value of the salary attribute of the record to its original 5000. As a result, however, the update of the other transaction, i.e., t4, is lost. Hence, t4 must also be backed up so that its update may be executed again. In other words, backing up of one transaction may cause another transaction to be backed up. In general, backing up of one transaction may cause several other transactions to be backed up. Hence, such back-ups are costly and time-consuming.

Our second motivation against transaction back-up can be illustrated with the following scenario. Let there be two back-ends in MDBS. Let us assume that the concurrency control at back-end 1 causes a transaction to back-up. The backed-up transaction at back-end 1 will give up its locks and start execution again from the beginning at a later time. Meanwhile, however, the same transaction is able to run to its completion at back-end 2 without any back-up. This may lead to two different (execution) sequences of the transactions requests at the two back-ends, thus causing a loss of monolithic consistency.

Thus, to be able to execute incompletely-specified transactions in MDBS, two alternatives are available to us. In one solution, we ensure that no backing up of transactions is required. In the second solution, we ensure that, even if backing-up is required, the two problems characterized above are eliminated. We will consider these two solutions in turn.

6.10.2 The No-Back-Up Solution

Before we may propose such a no-back-up solution, we repeat the cause for backing up transactions as illustrated in the previous example. There, a transaction has to be backed up because the transaction contains a request being executed which is non-permutable with a request in an earlier transaction. The occurrence of this situation is due to the fact that the earlier transaction is not completely specified and the non-permutable update of the earlier transaction is only received after the later transaction has begun the execution of its update request. The solution now becomes obvious. Before we begin to execute an incompletely specified-transaction, we must ensure that all earlier transactions are completely specified. Thus, if we allow for incompletely-specified transactions in MDBS, we will execute an incompletely-specified transaction only if all earlier transactions are completely specified. This is our first solution to the problem of handling incompletely-specified transactions.

6.10.3 A Solution with Backing Up

The second solution for executing incompletely-specified transactions is as follows. We begin to execute incompletely-specified transactions even when all earlier transactions are not completely specified. This, as we know may lead to transaction back-up. There are two problems caused by transaction back-up. First, when one transaction is backed up, other transactions may also need to be backed up. This happens for the following reason. Consider that the transaction t1 updates a record R1 in cluster 1. After the update, t1 releases the lock on cluster 1. This lock is subsequently acquired by the transaction t2 which also updates R1 in cluster 1. Later on, t1 needs to be backed up and the new values of the attributes in R1 have to be restored to their original values prior to that update by t1. This causes the update of t2 to be lost. Hence, t2 also needs to be backed up. In other words, backing up of t1 causes backing up of t2. This happens only because t1 has released the lock on cluster 1 before t1 completes its transaction. This lock released by t1 is subsequently acquired by t2. The problem would not arise if t1 holds on to all its locks until the end of transaction. In the terminology of concurrency control, the transactions must follow the two-phase locking protocol. This takes care of the first problem associated with transaction back-up.

The second problem has to do with the fact that a transaction may be backed up at one back-end and not at another. Hence, the transactions are executed in different orders in the different back-ends and this leads to a loss of monolithic consistency. One approach is to ensure that if a transaction is backed up at one back-end, it is also backed up at all back-ends. This may be achieved by exchanging control messages among the various back-ends. However, we wish to alleviate the control message traffic problem in MDBS and, hence, we reject this approach. The other approach is to ensure that even if a transaction has to be backed up at one back-end, the transaction retains the same position in the order of execution at all the back-ends. This may be achieved by ensuring that all transactions follow the two-phase protocol and hold all their locks until the very end of execution of the transaction. Then, if they have to be backed up in the middle of execution, they still have all their locks and, hence, they will maintain their position in the order of execution.

To summarize, both problems related to transaction back-up may be overcome by ensuring that all transactions follow the two-phase protocol and hold all their locks until the very end of execution of the transaction. The process of transaction execution in MDBS when incompletely-specified transactions are allowed and when the solution with backing up procedure is adopted is as follows.

At the beginning, a transaction will lock all the clusters it needs (this set of clusters is only incompletely specified at this time) in the to-be-used category and the appropriate mode (see Section 6.8.2). Now, the transaction will have its requests executed one by one. Before a request in a transaction may be executed, the appropriate locks have to be converted from the to-be-used to the being-used category. The process of lock conversion has already been explained before. After the appropriate locks have been converted, the request may then be executed. However, after the execution of the request, the locks used by the request are <u>not</u> released. They will not be considered for release until the transaction becomes completely specified and all requests in the transaction are executed. Even then, the locks are not released until all earlier transactions have been completely specified and executed and have released all their locks. Only then, does this transaction release all its locks.

We now explain what happens when a new request of an incompletely-specified transaction is received. The request is executed until the set of clusters needed by the request is determined. This set is used to update the set of clusters needed by the transaction. If the new request is non-permutable with a previously executed request from a later transaction, the later transaction has to be backed up and re-executed. The back-up is achieved by resetting the values of attributes in records updated by the transaction to their original values.

This completes our description of the second solution for executing incompletely-specified transactions in MDBS. We wish to emphasize that, in the above solution, the need for transaction back-up is detected the moment a non-permutable request from an earlier transaction is received. This is in contrast to the solutions of [Dewi80] and System D where the need for transaction back-up is not detected until the very end of a transaction. Together with the one in the previous section, we have suggested two solutions for MDBS to execute incompletely-specified transactions. Either solution may be employed in MDBS.

7. DESIGN AND PERFORMANCE ANALYSIS

In Chapter 1, we stated that we would propose the design of a multiple back-end system in which the ideal goal of its response time being proportional inversely to the multiplicity of its back-ends may be achieved. In the preceding five chapters, we revealed the design of such a system, known as MDBS. We now determine how well the ideal goal has been achieved by the MDBS design.

There are two approaches to the determination of the MDBS design in meeting the ideal goal. One may use analytic models based on the queueing theory to analyze the flow of information in MDBS and to measure the designed features of MDBS. One may also use simulation techniques to analyze and measure the behavior of MDBS.

We follow the analytic approach in Chapter 4 for selecting an appropriate strategy for directory management and request execution. As a result of the analytic modelling used in Chapter 4, we conclude that the ideal goal has been achieved and may even been surpassed by the MDBS design. However, a number of shortcomings of that analytic study is cited here. First, only retrieve requests were modelled and insert, delete and update requests were not modelled in that study. Second, the analytic model used is a closed queueing network model in which the total number of requests in the system is fixed. The modelling of MDBS as a closed system is valid as long as all the users of MDBS are issuing requests from the terminals. However, to take care of users who may submit requests as background transactions, we would also like to model MDBS as an open system in which the total number of requests in the system is not fixed but is dependent on the arrival rate of requests and the speed of MDBS in processing these requests. This is not done in the analytic study of Chapter 4. The third shortcoming of the analytic study is that many of the finer design details of MDBS can not be modelled. For instance, the concept of clusters and the placement strategy employed in MDBS are not modelled in the analytic study of MDBS in Chapter 4. Finally, the volume of the MDBS design and the limitation of the queueing theory render the use of either the closed or the open queueing network model impossible for getting meaningful theoretical results on the finer design and performance details of MDBS. Thus, we decided to employ simulation techniques in order to overcome the shortcomings of the analytic study of Chapter 4.

[illegible] organization of the rest of this chapter is as follows. In Section 7.1, [illegible] simulation model of MDBS. In Section 7.2, we present a measure of performance and the parameters of our simulation model. In Section 7.3, we represent our results and interpretation regarding the design and performance of MDBS.

7.1 A Simulation Model of MDBS

As we know, MDBS consists of a controller attached to a number of back-ends via a time-shared bus to which we refer as the broadcast bus (see Figure 7 again). In our simulation model, we assume that the controller is a VAX-11 computer and that the back-ends are PDP-11/44 minicomputers. Furthermore, the disk drives in our simulation model are assumed to have the characteristics of the RM02 disk drive. This characterization is realistic since the proposed MDBS is being implemented on a VAX-11 controller and PDP-11/44 back-ends with RM02 disk drives. Thus, our simulation study can be used to predict the design and performance of the system being implemented. Conversely, the result of our simulation study of MDBS design and performance can be verified by the actual performance of the implemented systems.

Our model simulates the sequence of events that takes place between the time that a user request enters MDBS and the time that the response data for the request is sent to the user. This sequence of events varies depending upon the type of the request. We describe, below, the sequence of events corresponding to each of the four request types in MDBS.

7.1.1 Sequence of Events for a Retrieve Request

The processing of the retrieval request takes place in several distinct phases as described below.

A. The Parsing Phase

When a request is scheduled for execution, the VAX-11 controller first parses the request and then broadcasts the request as a message to all the back-ends. The message goes to all the back-ends first via the VAX Unibus and then via the broadcast bus (i.e., DEC's PCL). It is assumed that both the VAX Unibus and DEC's PCL are reliable and guarantee the delivery of all messages in the same order that they are presented to them. In our model, we simulate first-in-first-out (fifo) queues at the VAX Unibus and at the broadcast bus. The time taken to transmit a message over the VAX Unibus or the DEC's PCL depends upon the size of the message.

B. The Descriptor Search Phase

Eventually, the broadcasted retrieve request is placed in a fifo queue at each of the back-ends. Each back-end processes its queue sequentially. On encountering this request in its queue, a back-end will perform descriptor processing for the request. That is, if in the request there are x predicates in the query and in MDBS there are n back-ends, each back-end will process x/n of the predicates and determine the corresponding descriptors of the predicates. In determining the corresponding descriptors, a back-end may need to access the secondary memory because the augmented descriptor-to-descriptor-id-table (DDIT) which maintains the descriptors may be in the secondary memory. In that case, an I/O request is generated by the back-end and placed in the queue associated with the disk drive which contains the augmented DDIT. Thus, in addition to a fifo queue for user requests, the back-end has a number of i/o queues.

Each disk drive has an i/o queue of access operations which have been issued by the back-end to which the disk drive is attached. The disk drives are assumed to use a first-come-first-served (fcfs) strategy in processing access operations in their queues. When a seek to a disk track is completed by a disk drive, the disk drive must wait until the unibus of the PDP-11/44 back-end to which it is attached becomes free before it may transmit the selected track to the back-end.

Eventually, the back-end will complete descriptor processing on the request. It will then broadcast the corresponding set of descriptors so determined to all the back-ends via the broadcast bus. For a request of n predicates, a back-end must complete the processing of its x/n predicates and receive the corresponding set of descriptors for the remaining (n-1)x/n predicates from the other (n-1) back-ends before the back-end is allowed to do any further processing on that request. However, in the meantime the back-end may process other requests in its fifo queue.

C. The Address Generation Phase Including Access Control

Consider that a back-end has finished the descriptor processing phase for a request and that it has received the remaining (n-1) corresponding sets of descriptors for this request from the other (n-1) back-ends. The back-end may then proceed to do address generation for that request by accessing and processing its augmented cluster definition table (CDT). At the end of this phase of processing, the addresses (of the records) of the authorized clusters for the user request are determined by the back-end. Originally, these addresses were generated for each cluster at the database-creation time in accordance with the track-splitting-with-random-placement strategy for placing the records of the cluster in the secondary storage.

D. The Secondary Memory Retrieval Phase

Finally, the back-ends access the tracks containing the records of the authorized set of clusters. Thus, the back-ends will generate a set of i/o operations, one for each track to be accessed, and place these i/o operations in the queues of the appropriate drives. As it receives tracks of records from the drives, the back-end processes these tracks by selecting those records which satisfy the user's query. Thus, the processing of tracks of records at the back-ends proceeds in parallel with the accessing of other tracks by the disk drives.

E. The Response Phase

Values of intended attributes are extracted from the selected records and sent over the broadcast bus and via the VAX-11 Unibus to the VAX-11 controller. When the entire response to the request has been received from all the back-ends, the controller outputs the response set to the user.

7.1.2 Sequence of Events for a Delete Request

The first four phases of a delete sequence are exactly the same as the ones of a retrieve sequence. However, the following phases are different.

A. The Tag-for-Deletion Phase

The selected records are marked with deletion tags. The marked records are then written back to the secondary store. In order for a back-end to write records with deletion tags onto the secondary store, the back-end must generate some i/o operations and place them in the fcfs queues of appropriate disk drives.

B. The Acknowledgement Phase

After the records have been properly written on to the secondary store, a message is sent to the controller via the broadcast bus and the VAX-11 Unibus, indicating the successful completion of the delete request. When such acknowledgement has been received from all the back-ends, the controller outputs a positive acknowledgement to the user.

7.1.3 Sequence of Events for an Update Request

As we recall, an update request in MDBS is associated with a modifier of type-0, type-I, type-II, type-III or type-IV. An update request with a modifier of type-III or type-IV is processed as a retrieve request followed by an update request of type-0. Thus, we simulate only update requests with modifiers of types-0, type-I and type-II.

The first four phases of an update sequence are exactly the same as the ones of a retrieve sequence. However, the remaining two phases are different.

A. The Record Modification and Cluster Calculation Phase

Each back-end must update the selected records by employing the modifier in the update request and calculate a new cluster number for each updated record. In order to keep the simulation of updates simple, we assumed that updated records do not change clusters. Rather, they continue to belong to the same cluster(s) and are inserted back into the same track(s) from which they are retrieved. Then, the updated records are written back to the secondary memory.

B. The Acknowledge Phase

After the update is successfully completed, the back-end sends a message to the controller via the broadcast bus and the VAX-11 Unibus, indicating completion of the update request. When such messages have been received from all the back-ends, the controller outputs a positive acknowledgement to the user.

7.1.4 The Sequence of Events for an Insert Request

As for the other three request types, the execution of an insert request also proceeds in several distinct phases.

A. The Parsing Phase

When an insert request is scheduled for execution, the VAX-11 controller will parse the request and then broadcast it to all the back-ends via the VAX Unibus and the broadcast bus.

B. The Descriptor Search and Initial Address Generation Phases

The request is now placed in the fifo queues of the n back-ends. When a back-end encounters this request in its queue, the back-end will find the corresponding descriptors for x/n of the total of x keywords present in the record for insertion. In order to find the corresponding descriptors, a back-end has to search the augmented DDIT. If the necessary descriptors are in the secondary memory, an i/o operation is generated by the back-end and placed in the fcfs queue associated with the disk drive which contains the descriptors in question.

After a back-end finds the corresponding descriptors for x/n of the directory keywords, the back-end will then broadcast its corresponding descriptors to all the other back-ends via the broadcast bus. Each back-end must complete the processing of the x/n directory keywords and receive the corresponding descriptors for the remaining (n-1)x/n directory keywords from the other (n-1) back-ends. Only then, can the back-end do further processing on the insert request. However, the back-end may process other requests in its queue while it is waiting for the corresponding descriptors to be broadcasted from the other back-ends.

After completing the descriptor processing on the insert request and receiving the (n-1) messages from the other back-ends, the back-end proceeds to the first step of address generation. In this step, it searches the augmented CDT to determine the cluster number of the record for insertion. Then, in the second step of address generation, it checks to see if the user that issued the insert request is allowed to make the insertion into the cluster determined in the first step of address generation. If not, the execution of the request terminates at this point. Otherwise, the back-end sends the cluster number to the controller via the broadcast bus and the VAX-11 Unibus.

C. The Back-end Selection Phase

The controller waits for messages from all the n back-ends. Then, it consults the cluster-id-to-next-back-end-table (CINBT) for selecting a next back-end for the record insertion. The controller will then send a message to this back-end via the VAX-11 Unibus and the broadcast bus. The message is placed in the fifo queue at the selected back-end.

D. The Record Insertion Phase

On encountering the insertion message in its queue, the back-end proceeds with the final step of address generation. It will first search the augmented CDT to determine the track into which the record is to be inserted and then generate an i/o operation to the disk drive containing the selected track. After the back-end receives a positive message from the disk drive, the back-end in turn sends a message to the controller via the broadcast bus and the VAX-11 Unibus.

E. The Acknowledgement Phase

Upon receiving a positive message from a single back-end, the controller acknowledges to the user that the record is successfully inserted.

7.2 Simulation Environments and A Measure of Performance

The simulation model described in the last section is programmed using Simula on a DEC System 20. Because the model is highly parameterized, the number of distinct cases that can be formulated is extremely large. The combinatorics inherent to such modelling makes any exhaustive simulation infeasible. Fortunately, a great deal can be learned by simulating an appropriately chosen subset of all the possible cases involved. See appendices H and I.

7.2.1 Retrieve-Intensive vs. Update-Intensive

Let the environment of MDBS in a particular application be defined by the percentages of the four request types that will be encountered in that application. Then, a retrieve-intensive environment is one in which a large percentage of requests received by MDBS are retrieve requests and in which very few delete, update and insert requests are received. Similarly, an update-intensive environment is one in which a large percentage of requests received by MDBS are update, delete and insert requests, and in which the percentage of retrieve requests encountered is low. Any environment may be accommodated by our simulation model. However, we choose to model two specific environments - a retrieve-intensive one in which 100% of the requests are retrieves and an update-intensive one in which there will be 25% of each of the four request types. Thus, we are modelling the two ends of the spectrum of possibilities for the environment of MDBS.

7.2.2 Cluster Size vs. Request Size

In our study, we assumed that a total of 10,000 clusters exist in the database and that these clusters are placed using the track-splitting-with-random-placement strategy described in Chapter 2. The average number of tracks in a cluster is chosen from the set {2,10}. In the case of update, delete and retrieve requests, we also need to choose the number of clusters that will be updated, deleted or retrieved by a request. We run tests for small-sized requests in which the number of clusters to be processed (i.e., to be updated, deleted or retrieved) varies between 1 and 20, and for large-sized requests in which the number of clusters to be processed varies between 20 and 40. The number of predicates in a query is chosen to be uniformly distributed in the range from 1 to 5.

7.2.3 Hardware Configurations and Requirements

The broadcast bus, i.e., DEC's PCL, is chosen to have a transmission speed of 1 mbytes/ sec and the VAX Unibus is chosen to have a transmission speed of 2 mbytes/ sec. It is assumed that both the controller and the back-ends must execute about 8,000 instructions, taking 8 msecs, in order to generate messages for broadcast. Furthermore, we assume that each message has to be augmented with 50 bytes of header information. In other words, the minimum possible size of a message in MDBS is 50 bytes. Finally, the number of back-ends is chosen from the set {3,6,9}.

7.2.4 A Measure of Anticipated Performance vs. Ideal Performance

In order to be able to present our results more effectively, we define a performance measure, called the percentage ideal goal. We take the response time of MDBS with three back-ends as our reference point. We then define the percentage ideal goal of MDBS utilizing n back-ends as the following ratio:

$$\frac{3 \times (\text{response time of MDBS with 3 back-ends}) \times 100}{n \times (\text{response time of MDBS with n back-ends})}$$

With the above definition, we learn that, the percentage ideal goal of MDBS utilizing six back-ends will be 100, only if the response time of MDBS with six back-ends is exactly one half of the response time of MDBS with three back-ends. Similarly, when the number of back-ends is nine, the percentage ideal goal of MDBS will be 100, only if the response time of MDBS with nine back-ends is exactly one third of the response time of MDBS with three back-ends. Thus, the percentage ideal goal of MDBS is a measure of how close MDBS is to achieveing its ideal performance goal.

7.3 MDBS Performances Under Various Conditions

On the basis of the measure, environments and parameters, we conduct the simulation of MDBS. In this section, we tabulate the simulation results and discuss these tables. In the discussion, we try to interpret the findings and relate the findings to the design details of the MDBS.

7.3.1 Intensive Retrieval Involving Large Clusters

From Figure 44, we see that the performance of MDBS is better than anticipated when the average number of tracks per cluster is chosen to be ten. In fact, the percentage ideal goal reaches as high as 140 for large-sized requests when the number of back-ends is nine and the interarrival time of requests is 3.5 sec. The reason for the percentage ideal goal being greater than 100 has been explained in Chapter 4 and has to do with the utilization of the disk system. We recall, when the number of back-ends in MDBS is doubled, the number of tracks to be accessed by each back-end is halved due to our data placement strategy. Furthermore, the time to access a track is less when more back-ends are used. Both these factors contribute to a response-time improvement which is better than 'ideal'.

7.3.2 Intensive Retrieval Involving Small Clusters

From Figure 45, we learn that the percentage ideal goal can be as low as 61.5 when the average number of tracks per cluster is two and the requests are of small-sized. The percentage ideal goal remains low, even the requests are of large-sized. In other words, the performance of MDBS deviates furthest from the ideal goal when the average number of tracks per cluster and the average number of clusters to be retrieved by a request are both very small. The interpretation is as follows. Our design has strived to alleviate the controller limitation problem and the control message traffic problem. However, neither of these two problems could be entirely eliminated in our design. Thus, a constant amount of time, independent of the number of back-ends, is spent by the controller to parse user requests, to broadcast user requests to all back-ends and to output results to the user. Similarly, the back-ends need to exchange messages in order to synchronize descriptor processing on a request. The effects of these tasks on the response time of a request (and, therefore, on the achievement of the ideal goal) will be negligible when the number of tracks to be retrieved from the secondary memory is large. This is because when a large

Average Number of Tracks per Cluster is 10.
Retrieve-intensive environment when all requests are retrieval requests.

Small-sized Requests (involving 1 to 20 clusters)						
# of back-ends / Inter-arrival time in sec.	3		6		9	
	Response time (sec)	% Ideal Goal	Response time (sec)	% Ideal Goal	Response time (sec)	% Ideal Goal
1.5	2.06	100	.84	122.6	.583	117.78
2	1.06	100	.786	105.6	.567	97.6

Large-sized Requests (involving 20 to 40 clusters)						
# of back-ends / Inter-arrival time in sec.	3		6		9	
	Response time (sec)	% Ideal Goal	Response time (sec)	% Ideal Goal	Response time (sec)	% Ideal Goal
3.5	4.53	100	1.66	136.44	1.1	140.3
5	3.16	100	1.51	104.64	1.05	100.3

Figure 44. The MDBS Response Times In a Retrieve Intensive Environment with Large Amount of Data Involved

The average number of tracks per cluster is 2
Retrieve-intensive environment where all requests are the retrieval requests.

Small-sized Requests (involving 1 to 20 clusters)						
# of back-ends / Inter-arrival time in sec.	3		6		9	
	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	.571	100	.364	78.4	.304	62.6
2	.55	100	.359	76.6	.298	61.5

Large-sized Requests (involving 20 to 40 clusters)						
# of back-ends / Inter-arrival time in sec	3		6		9	
	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	1.04	100	.586	88.7	.458	75.7
2	.957	100	.567	84.4	.445	70

Figure 45. The MDBS Response Times in a Retrieve-Intensive Environment with Small Amount of Data Involvement

fraction of the response time will be spent in accessing the secondary memory, the effects of these tasks on the overall response time is relatively low. On the other hand, if the number of tracks to be accessed is very small, then the overall response time to a request is also very small. In this case, the effect, of these tasks on the response time, will result in relatively heavy impact. This is why the performance of MDBS deviates furthest from the ideal goal when the average number cf tracks in a cluster and the average number of clusters to be retrieved for a request are both very low.

7.3.3 Intensive Update Involving Large Clusters

The results tabulated in Figure 46 for an update-intensive environment are similar to those of Figure 44. Thus, when the average number of tracks per cluster is ten, the percentage ideal goal can reach as high as 190. This happens for large-sized requests when the number of back-ends is nine and the inter-arrival time of requests is 3.5 sec.

In comparing this figure (i.e., Fig. 46) with Figure 44, we note that the highest percentages achieved are 190 and 140, respectively. One may wonder why would the update-intensive requests achieve higher percentage (i.e., 190) than the retrieve-intensive requests under the same condition, since updates tend to tax the system performance more pronouncedly than retrieves do. It turns out that these two percentages are not directly related to each other. What they have indicated is that with more back-ends the slow update (with a response time of over six seconds) in a 3-back-ends setting may be speeded up more dramatically where as the fast retrieval (with a response time under five seconds) in a 3-back-end setting may be speeded up not as dramatically as the updates. By the time MDBS is a system of 9 back-ends, the response times of updates and retrievals are both close to one second as depicted in Figures 46 and 44, respectively.

7.3.4 Intensive Update Involving Small Clusters

For small values of the average number of tracks in a cluster and the average number of clusters to be accessed in a update request, the percentage ideal goal can get as low as 60. Comparing corresponding entries in Figures 45 and 47, we see that the performance of MDBS is closer to ideal in the update-intensive environment than in the retrieve-intensive environment.

The reason for this is as follows. Update and delete requests take longer

The Average number of tracks per cluster is 10.
The Update-intensive environment consists of 25% Inserts, 25% Deletes, 25% Updates, 25% Retrieves.

Small-sized Requests (1 to 20 clusters)						
# of back-ends	3		6		9	
Inter-arrival time in sec	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	3.1	100	1.03	150.4	.702	147
2	2.24	100	.946	118.4	.667	112

Large-Sized Requests (20 to 40 clusters)						
# of back-ends	3		6		9	
Inter-arrival time in sec	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
3.5	6.73	100	2.24	150.29	1.18	190
5	5.67	100	2.08	136.3	1.18	160.17

Figure 46. The MDBS Response Times in an Update-Intensive Environment Involving Large Clusters

The average number of tracks per cluster is 2.
The update-intensive environment where there are 25% insert, 25% delete, 25% update and 25% retrieve requests.

Small-sized Requests (1 to 20 clusters)						
# of back-ends / Inter-arrival time in sec	3		6		9	
	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	.604	100	.398	76	.335	60
2	.581	100	.391	74.3	.329	59

Large-sized Requests (20 to 40 clusters)						
# of back-ends / Inter-arrival time in sec	3		6		9	
	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	1.530	100	.759	100.7	.565	90.2
2	1.260	100	.702	89.7	.526	80

Figure 47. Intensive Update Involving Small Clusters

AD-A104 445 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
DESIGN AND ANALYSIS OF A MULTI-BACKEND DATABASE SYSTEM FOR PERF--ETC(U)
AUG 81 M J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-81-8 NL

2 OF 2
AD
A104445

END
DATE
FILMED
10-81
DTIC

to execute than retrieve requests because, besides requiring the retrieval of data from the secondary memory, the former requires the insertion of tracks containing the updated and deleted records in to the secondary memory. As a result, the average response time of MDBS in an update-intensive environment is greater than in a retrieve-intensive environment. Hence, the overall response time in an update-intensive environment will be less affected by the time for the fixed 'overhead' tasks such as request parsing and acknowledgement than the overall response time in a retrievel-intensive environment.

7.3.5 Effects of Broadcasting on Performance

It is easy to see that for a given request, the amount of data which is sent to the controller from the back-ends and which is output by the controller to the user is a constant, independent of the number of back-ends. As a result, the time taken to transmit this data over the broadcast bus is a part of the overall response time of MDBS which cannot be decreased by increasing the number of back-ends. Thus, a slow broadcast bus may cause the performance of MDBS to deviate away from the ideal goal. In this section, we propose to investigate the effects of broadcast bus speed on the performance of MDBS.

In previous sections, we presented results on the MDBS response time with the assumption that the broadcast bus could transmit at a rate of 1 mbytes/sec. We now present two more sets of results, one for the case when the broadcast bus can transmit at 0.5 mbytes/sec and the other for the case when the broadcast bus can transmit at 2 mbytes/sec. Since the deviation of MDBS performance away from ideal is greatest for small number of tracks per cluster and small-sized requests, we present results only for this case. Furthermore, since the effects of a slow bus will be felt the most in a retrieve-intensive environment where large amounts of data have to be returned over the broadcast bus, we will only simulate such an environment. The results are tabulated in Figure 48.

The results indicate that as long as the broadcast bus can transmit at a speed greater than 0.5 mbytes/sec, the overall response time of MDBS is unaffected. Note that these results should be read in conjunction with the results in Figure 45 for the case where the broadcast bus can transmit at a speed of 1 mbyte/sec. We conclude that the speed of the broadcast bus is not likely to affect the performance of MDBS in its achievement of the ideal goal, since buses with speeds of greater than 0.5 mbytes/sec are commercially available.

The average number of tracks per cluster is 2.
The environment is Retrieve-intensive with small-sized requests
(1 to 20 clusters)

Bus speed = 0.5 Mbytes/sec						
# of back-ends	3		6		9	
Inter-arrival time in sec	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	.571	100	.364	78.4	.304	62.6
2	.55	100	.359	76.6	.298	61.5

Bus speed = 2 Mbytes/sec						
# of back-ends	3		6		9	
Inter-arrival time in sec	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	.571	100	.364	78.4	.304	62.6
2	.55	100	.359	76.6	.298	61.5

Figure 48. The Response Times of MDBS Effected by the Broadcast Bus Speeds

An example of one such bus is the PCL-11 bus [Uhri79] provided by Digital Equipment Corporation.

7.3.6 Three Observations of Strong Design and Performance Factors - High-Volume Processing, Intensive Update and Inexpensive Broadcast Bus

Let us summarize our results up to this point. MDBS will achieve and surpass its ideal goal as long as the requests issued to it are such that large amounts of data have to be read and manipulated in order to process them. This is precisely the kind of environment for which MDBS has been designed. On the other hand, the ideal goal of MDBS may be reached by only 60% if requests are such that only very small amounts of data have to be read and manipulated in order to process them.

Secondly, our results have shown us that MDBS will achieve its ideal goal more closely in an update-intensive environment than in a retrieve-intensive environment because of the additional accesses to the secondary memory needed in the former case.

Finally, our results have shown us that the speed of the broadcast bus is not a bottleneck to its ideal performance. As long as the speed of this bus is greater than 0.5 mbytes/sec, the performance of MDBS is unaffected by the speed of the broadcast bus.

7.4 A More Refined Simulation of MDBS

In the simulation experiments of the previous section, we had assumed that the controller broadcasts a request to all the n back-ends in MDBS. Actually, the controller will need to broadcast a request to only x back-ends, where x is the number of predicates in a retrieve, delete or update request or the number of keywords in an insert request, if x is less than n. By use of such a policy, the synchronization overhead in MDBS is reduced. Thus, during the descriptor processing phase, each back-end will need to wait for results from only (x-1) rather than (n-1) other back-ends. Furthermore, such a policy will result in fewer message exchanges and lesser traffic on the broadcast bus. This refinement is now incorporated into the simulation model of MDBS.

The response times of MDBS under the refined policy are tabulated in Figure 49. The corresponding results for MDBS without the refined policy are the ones in Figure 45. We only simulate MDBS with the refined policy when the average number of tracks per cluster is small, i.e., two. This is because the

The average number of tracks per cluster is 2
The environment is Retrieve-intensive

Small-sized Requests (1 to 20 clusters)						
# of back-ends / Inter-arrival time in sec	3		6		9	
	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	.521	100	.286	91.08	.193	90.0
2	.507	100	.283	89.5	.196	86.05

Large-sized Requests (20 to 40 clusters)						
# of back-ends / Inter-arrival time in sec	3		6		9	
	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal	Response time in sec	% Ideal Goal
1.5	.104	100	.466	111.58	.304	114.03
2	.915	100	.454	100.77	.306	99.67

Figure 49. The Response Times of MDBS Under a Refined Policy Simulation

performance of MDBS is ideal or better when the average number of tracks per cluster is ten even with no use of the refined policy. Thus, we are only interested in seeing if the refined policy can improve the performance of MDBS in the region where it is operating below ideal. Comparing corresponding entries in Figures 45 and 49, it is clear that MDBS response times under the refined policy are lower than those without such a policy. However, more importantly, the MDBS performance is now much closer to ideal. Thus, in the worst case, the percentage ideal goal of MDBS is 86 as opposed to 60 where the refined policy is not employed. We do not believe that MDBS performance can be any closer to ideal because the controller limitation problem and the control message traffic problem can never be completely eliminated.

8. DESIGN GOALS AND ACHIEVEMENTS - A SUMMARY AND REVIEW

It is generally known that the use of a single general-purpose digital computer with dedicated software for database management as a back-end to off-load the mainframe host computer from database management tasks yields no appreciable gains in performance. Furthermore, to replace the back-end computer hardware and software with more powerful hardware and newly designed software may be costly and disruptive. Such an upgrade would require major efforts since these single back-end database management systems are not designed for hardware extension and software enhancement.

In this dissertation, we first make the claim that for the management of very large databases, the use of multiple minicomputers in a parallel fashion may be feasible and desirable. By feasible we mean that it is possible to configure a number of back-end minicomputers each of which is driven by identical database management software and controlled by a controller (minicomputer) for concurrent operations on the database spread over the disk storage local to the back-end minicomputers. This approach to large databases may be desirable because only off-the-shelf equipment of the same kind is utilized to achieve high performance without requiring specially-built hardware and because identical database management software may be replicated on the back-ends. This approach makes the capacity growth and performance improvement easy because duplicate hardware can be added and used with replicable software.

We then present in this dissertation a new approach to the solution of database management problems involving database growth and performance improvement. The approach utilizes a multiplicity of conventional minicomputers, a novel hardware configuration and an innovative software design. This extensible system, called MDBS, is designed to achieve the ideal goal of having the response time of MDBS to be proportional inversely to the multiplicity of back-ends.

In presenting our approach, our initial effort is to identify the problems and bottlenecks involved in designing such an ideal system. These problems are identified by surveying and examining existing software-oriented multiple back-end systems. In order to overcome the problems identified and in order to develop an ideal system, we set nine design goals for MDBS.

First, we resolve to eliminate the channel limitation problem. That is, we do not wish the throughput of MDBS to be limited by the transfer rate of data from the secondary store via i/o channels. Second, we resolve to alleviate the controller limitation problem by executing all the database management functions in a parallel fashion in the back-ends rather than at the controller. For our third design goal, we decide that all back-ends must execute identical software. As a result, capacity growth and performance improvement with the use of additional back-ends and replicable software becomes a straightforward task. Fourth, we resolve to minimize communications among the back-ends and between the back-ends and the controller. Without excessive communications, the performance of MDBS will not taper off after the first few additional back-ends. Fifth, we decide not to use any special-purpose hardware in MDBS. This makes the addition of duplicate hardware for capacity growth and performance improvement easy. It also makes the replication of software on additional hardware easy. Sixth, we decide that each back-end should support concurrent request execution for better resource utilization and system response time. For our seventh goal, we resolve that more than one disk drive would be attached to each back-end. As a result, very large databases of the magnitude of, say, 10^{10} bytes may now be supprted in MDBS. Eighth, we decide to design MDBS in such a way that all back-ends would participate equally in the execution of a request. This is essential for achieving an ideal system in which the response time is proportional inversely to the number of back-ends. Finally, for our ninth goal, we resolve to find a canonical data model into which prevailing data models such as the relational, network and hierarchical models can be translated in a straightforward manner.

We begin our design of MDBS by specifying that it would consist of a controller and several back-ends. Each back-end is attached with several disk drives which store the database. As a result of attaching several disk drives to each back-end, we are able to achieve our seventh goal. Furthermore, we are also able to overcome the channel limitation problem and, hence, to achieve our first goal. Thus, the throughput of MDBS is no longer limited by the transfer rate of data from the secondary storage via I/O channels. By using off-the-shelf equipment for the controller, the back-ends and the disk drives, we are able to achieve our fifth goal of not utilizing any special-purpose hardware in MDBS. Thus, we achieve three of our nine design goals.

We reach the second goal, that none of the major database management tasks is performed in the controller and all such tasks are accomplished in a parallel fashion by the back-ends, by the careful design of algorithms for request execution. Before it may access the database for the purpose of carrying out a request. MDBS must first access auxilliary information about the database. That is, MDBS must perform directory management for the request. A number of alternative strategies for directory management are proposed and evaluated by using a closed queueing network model. The new strategy chosen for directory management minimizes the overall response time of MDBS and requires minimal work to be performed at the controller of MDBS. Next, during the execution of a request, MDBS enforces access control. That is, MDBS performs access-control-related directory management. The proposed scheme for controlling access is also performed at the back-ends and is not done in the controller. In order to improve the response time of a request even further, it is necessary to have a mechanism which allows for concurrent execution of multiple requests. Such a mechanism is referred to as a concurrency control mechanism. The newly proposed concurrency control mechanism for MDBS also executes in the back-ends and not at the controller. As a result, every phase of request execution is performed in the back-ends and requires minimal work at the controller. Thus, we manage to alleviate the controller limitation problem and achieve our second goal in MDBS.

Every one of the algorithms mentioned above for directory management, for access control and for concurrency control in the course of request execution requires that identical software be executed in the back-ends. As a result, we achieve the third goal of MDBS to have replicable software in the back-ends to facilitate capacity growth and performance improvement with additional back-ends.

Our fourth goal is to minimize communications among the back-ends and between the back-ends and the controller. This is achieved by a number of different techniques. First, the controller and the back-ends are connected by means of a broadcast bus. As a result, the controller will need to use only a single message rather than n different messages in order to send a request to all the n back-ends. Simulation experiments show that the reduced message traffic caused by use of the broadcast capability can lead to improvements in the response time by a factor of as much as five. Communications traffic is also reduced by ensuring that the concurrency control mechanism in MDBS requires no exchange of messages among the back-ends, unlike all other concurrency control mechanisms for multiple back-ends. Finally, the communications traffic is reduced by the use of a data placement strategy which ensures that each back-end is to access the same amount of data as any other back-end in order to respond to a request. Thus, a back-end does not need to communicate with other back-ends or with the controller in order to retrieve data from the database. These techniques allow us to achieve the fourth goal of alleviating the control message traffic problem in MDBS.

Our sixth goal is to ensure that each back-end in MDBS supports concurrent request execution for the better resource utilization and system response time. We first argue that such concurrent request execution is beneficial for MDBS. Next, we determine the necessary and sufficient conditions for a consistent MDBS database that utilizes multiple back-ends. As a result, we develop the very important notion of monolithic consistency. A new algorithm for ensuring monolithic consistency is then described. Our algorithm is unique in a number of ways. First, it advocates the use of four lock modes, instead of the traditional two lock modes. By separating the insert and delete locks from the update locks, we achieve a greater degree of concurrency. Another contribution is the identification of permutable and compatible requests for a high-level query language such as MDBS' DML. At a practical level, a method for enforcing locking is proposed when a predicate-based query language is utilized. Unlike [Eswa76] which uses predicate locking, our scheme uses cluster locking and is simpler. Unlike [Jord81], our scheme allows predicate-based updates. Unlike both [Eswa76] and [Jord81], our scheme is deadlock-free. Hence, it cannot suffer from the so-called starvation problem where transactions are rolled back infinitely and are not guaranteed to complete. Thus we have achieved the sixth goal of MDBS.

Our eighth goal is to ensure that each back-end participates equally in the execution of a request. This is achieved by partitioning the database into equivalence classes which are termed clusters. Every record in the database belongs to one and only one cluster. In order to form clusters, the database-creator specifies descriptors. By proper use of descriptors, the clusters are formed in such a way that if a user needs to access a record belonging to a cluster, the user is most likely to have the need to access all the other records belonging to that cluster. Thus, clusters serve as the basic units of access in MDBS. In other words, every user request requires the retrieval of one or more clusters. By storing the clusters in such a way that each cluster is evenly distributed among the back-ends, we may ensure that every user request will require the retrieval of the same amount of data from all the back-ends. Thus, the record clustering and cluster placement algorithms ensure that each back-end does an equal share of data retrieval for a request. The directory management, access control and concurrency control algorithms are also designed so that each back-end performs an equal share of the work. Thus, we have achieved our eighth goal in MDBS.

Finally, for our ninth goal, we resolve to find a canonical data model into which prevailing data models such as the relational, network and hierarchical data models can be translated in a straightforward manner. In our quest for a canonical data model, we evaluated a number of data models on the basis of three criteria - the translation criterion, the partition criterion and the language criterion. We were able to show that the attribute-based model was the only one that satisfied all three criteria. Accordingly, we choose to implement the attribute-based model directly, and the other data models by translation, in MDBS. We also present a simple data manipulation language DML based on this data model. Thus, we have achieved our ninth and final goal for MDBS.

In this way, every one of the nine design goals set for MDBS is achieved. We believe that the resulting architecture comes close to achieving the ideal goal of having the response time be proportional inversely to the number of back-ends. In order to test our conjecture, a complete simulation model of MDBS is designed on a DEC System 20 using Simula. Several simulation experiments are run on this model.

The results of the simulation experiments prove to be very satisfactory. It is seen that the MDBS response time is ideal or better under typically expected conditions when the number of tracks to be accessed and processed in order to satisfy a user request is large. The reason for the MDBS response time being better than ideal in some cases is closely related to the utilization of the disks which store the database. When the number of back-ends in MDBS is increased, not only is the number of tracks to be accessed at each back-end decreased, so is the time to access each track. This explains the reason for the better-than-expected decrease in the response time.

The results of the simulation experiments also show that the MDBS response time is not ideal when the number of tracks to be accessed and processed is small. However, the deviation from the ideal response time is very slight and is never more than 20%. The reason for the less-than-ideal performance is due to the controller limitation problem and the problem of control message traffic which cannot be entirely eliminated but can only be alleviated. More specifically, there are four tasks which are to be executed in MDBS where the execution time of these tasks is independent of the number of back-ends in MDBS.

First, there is the parsing task performed in the controller. That is, the controller has to spend a certain amount of time performing the parsing of a user request and this time is independent of the number of back-ends in MDBS. Second, there is the broadcasting task which is also performed by the controller and requires an amount of time independent of the number of back-ends in MDBS. For this task, the controller broadcasts a request to all the back-ends. Third, there is the outputting task. That is, the controller has to spend a certain amount of time, once again independent of the number of back-ends, in outputting the results of a request to the user that issued the request. The final task is the address generation task which is performed at each back-end. Under ideal conditions, a back-end must perform no address generation if none of the clusters to be retrieved is stored at that back-end. This is not achieved in our system. Thus, even when a back-end does not contain any of the required clusters, the back-end must still do address generation in order to discover that it does not contain any of the required clusters for a request. However, the execution of this task is carefully designed so that each back-end performs less work when the number of back-ends is increased.

Our simulation indicates that due to the fixed overhead of these four tasks the ideal goal set for MDBS is not being achieved when the number of tracks to be accessed and processed is very small. Nevertheless, we do not believe that the latter three tasks may be performed in any different way to improve the performance of MDBS. In other words, the latter three tasks are inherent to multiple back-end systems and are not a defect of MDBS. However, the performance of the first task may be improved, if we can come up with a parallel algorithm for parsing a user request. However, unless the user request is complex, the need for a parallel parsing algorithm for performing enhancement will be unnecessary. On the other hand, for requests involving large amounts of data, the fixed overhead incurred from the aforementioned tasks becomes negligible.

In conclusion, we believe that this dissertation has met its objectives in the design and analysis of a multiple back-end database management system for capacity growth, performance improvement and functionality enhancement.

REFERENCES

[Adabyy] ADABAS Reference Manual, Software AG, Reston, Virginia.

[Astr75] Astrahan, M.M., and Chamberlin, D.D., "Implementation of a Structured English Query Language," CACM, Vol. 18, No. 10, October 1975, pp. 580-587

[Astr76] Astrahan, M.M., et. al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 189-222.

[Auer80] Auer, H., "RDBM - A Relational Data Base Machine," Technical Report No. 8005, University of Braunschweig, June 1980.

[Babb79] Babb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979, pp. 1-29.

[Bane77] Banerjee, J., Hsiao, D.K. and Kerr, D.S., "DBC Software Requirements for Supporting Network Databases," Technical Report, OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, June 1977.

[Bane78] Banerjee, J. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 3, No. 4, December 1978, pp. 347-384. Also available in Baum, R.I., Hsiao, D.K. and Kannan, K., "The Architecture of a Database Computer -- Part I: Concepts and Capabilities," Technical Report OSU-CISRC-TR-76-1, The Ohio State University, Columbus, Ohio, September 1976.

[Bane79] Banerjee, J., Hsiao, D.K. and Kannan, K., "DBC - A Database Computer for Very Large Databases," IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 414-429.

[Bane80] Banerjee, J., Hsiao, D.K. and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," IEEE Transactions on Software Engineering, March 1980; Also available in Hsiao, D.K., Kerr, D.S. and Ng, F.K., "DBC Software Requirements for Supporting Hierarchical Databases," Technical Report OSU-CISRC-TR-77-1, The Ohio State University, Columbus, Ohio, April 1977.

[Bard81] Bard, Y., "A Model of Shared DASD and Multipathing," CACM, Vol. 23, No. 10, October 1980, pp. 564-572.

[Baye76] Bayer, R., and Metzger, J.K., "On the Encipherment of Search Trees and Random Access Files," ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 37-52.

[Baye80] Bayer, R., et. al.,"Parallelism and Recovery in Database Systems," ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980.

[Bent75] Bentley, J.L., "Multidimensional Binary Search Trees Used for Associative Searching," CACM, September 1975, Vol. 18, No. 9, pp. 509-517.

[Bora80] Boral, H., et. al.,"Parallel Algorithms for the Execution of Relational Database Operations," Computer Sciences Technical Report No. 402, University of Wisconsin-Madison, October 1980.

[Bora81] Boral, H. and DeWitt, D.J., "Processor Allocation Strategies for Multiprocessor Database Machines," ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, pp. 227-265.

[Cana74] Canaday, R.H., et al, "A Back-End Computer for Data Base Management," CACM, Vol. 17, No. 10, October 1974, pp. 575-582.

[Card75] Cardenas, A.F., "Analysis and Performance of Inverted Data Base Structures," CACM, Vol. 18, No. 5, May 1975, pp. 253-263.

[Cham74] Chamberlin, D.D. and Boyce, R.F., "A Deadlock-Free Scheme for Resource Locking in a Database Environment," IFIP, August 1974, pp. 340-343.

[Cham75] Chamberlin, D.D., Gray, J.J. and Traiger, I.L., "Views, Authorization and Locking in a Relational Data Base System," Proceedings of the National Computer Conference, 1975, pp. 425-430.

[Chan80a] Chang, J.M. and Fu, K.S., "A Dynamic Clustering Technique for Physical Database Design," Proceedings of the ACM SIGMOD Conference on Management of Data, Santa Monica, California, May 14-16, 1980.

[Chan80b] Chang, C.C., Lee, R.C.T. and Du, H.C., "Some Properties of Cartesian Product Files," Proceedings of the ACM SIGMOD Conference on Management of Data, Santa Monica, California, May 14-16, 1980.

[Chu 78] Chu, W.W., Lee, D. and Iffla, B., "A Distributed Processing System for Naval Data Communication Networks," AFIPS Conference Proceedings, Vol. 47, 1978, pp. 783-793.

[Chu 80] Chu, W.W., et. al.,"Task Allocation in Distributed Data Processing," Computer Magazine, Vol. 13, No. 11, November 1980, pp. 57-69.

[Come79] Comer, D., "The Ubiquitous B-Tree," ACM Computing Surveys, Vol. 11, No. 2, June 1979.

[Cope73] Copeland, C.P., Lipovski, G.J. and Su, S.Y.W., "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Proceedings of the First Annual Symposium on Computer Architecture, December 1973, pp. 121-128.

[Cosm75] Cosmetalos, G.P., "Approximate Explicit Formula for the Average Queueing Time in the Processes M/D/r and D/M/r," Information, Vol. 13, October 1975.

[Datayy] Data Management System (DMS 1100), Amercian National Standard COBOL Data Manipulation Language, Programmer Reference UP-7908, Sperry Univac Computer Systems, St. Paul, Minn.

[Date75] Date, C.J., "An Introduction to Data Base Systems," Addison-Wesley, Reading, Mass., 1975.

[Denn78] Denning, P.J. and Buzen, J.P., "The Operational Analysis of Queueing Network Models," Computing Surveys, Vol. 10, No. 3, September 1978, pp. 225-261.

[Dewi78] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems," Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978.

[Dewi80] DeWitt, D.J. and Wilkinson, K.K., "Database Concurrency Control in Local Broadcast Networks," Computer Sciences Technical Report 396, University of Wisconsin-Madison, August 1980.

[Down77] Downs, D. and Popek, G.J., "A Kernel Design for a Secure Data Base Management System," Proceedings of the Conference on Very Large Databases, Tokyo, Japan, October 6-8, 1977, pp. 507-514.

[Epst78] Epstein, R., Stonebraker, M. and Wong, E., "Distributed Query Processing in a Relational Data Base System," Memorandum No. UCB/ERL M78/18, Electronics Research Laboratory, University of California, Berkeley, April 1978.

[Epst80] Epstein, R. and Hawthorn, P., "Design Decisions for the Intelligent Database Machine," Proceedings of the National Computer Conference, AFIPS, Vol. 49, 1980, pp. 237-241.

[Eswa76] Eswaran, K.P., et. al., "The Notions of Consistency and Predicate Locks in a Database System," CACM, Vol. 19, No. 11, November 1976, pp. 624-633.

[Fern75] Fernandez, E.B., et. al., "An Authorization Model for a Shared Data Base," Proceedings of the ACM-SIGMOD Conference on Management of Data, San Jose, California, pp. 23-31, May 1975.

[Fran74] Franklin, M.A. and Sen, A., "An Analytic Response Time Model for Single and Dual-Density Disk Systems," IEEE Transactions on Computers, Vol. C-23, No. 12, December 1974.

[Fran77] Franta, W.R., "The Process View of Simulation," Computer Science Library, North Holland, 1977.

[Gard77] Gardarin, G. and Lebeux, P., "Scheduling Algorithms for Avoiding Inconsistency in Large Databases," Third International Conference on VLDB, Japan, October, 1977.

[Good80] Goodman, J.R. and Despain, A.M., "A Study of the Interconnection of Multiple Processors in a Data Base Environment," International Conference on Parallel Processing, 1980, pp. 269-278.

[Gotl73] Goelieb, C.C. and Macewen, G.H., "Performance of Movable-Head Disk Storage Devices," JACM, Vol. 20, No. 4, October 1973, pp. 604-623.

[Gray78] Gray, J., "Notes on Data Base Operating Systems," Operating Systems, Lecture Notes in Computer Science, Vol. 60, Springer-Verlag, 1978.

[Grif76] Griffiths, P.P. and Wade, B.W., "An Authorization Mechanism for a Relational Database System," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 242-255.

[Hawt79] Hawthorn, P. and Stonebraker, M., "Performance Analysis of a Relational Database Management System," Proceedings of the ACM SIGMOD Conference on Management of Data, Boston, May 30 - June 1, 1979.

[Hawt80] Hawthorn, P. and DeWitt, D.J., "Performance Analysis of Alternative Database Machine Architectures," Computer Sciences Technical Report No. 383, University of Wisconsin-Madison, March 1980.

[Hawt81] Hawthorn, P., "The Effect of Target Applications on the Design of Database Machines," Proceedings of the ACM SIGMOD Conference on Mangement of Data, April 29 - May 1, 1981, pp. 188-197, Ann Arbor, Michigan.

[Hsia70] Hsiao, D.K. and Harary, F.A., "A Formal System for Information Retrieval from Files," CACM, Vol. 13, No. 2, February 1970, pp. 67-73.

[Hsia79a] Hsiao, D.K., Kerr, D.S. and Madnick, S.E., "Computer Security, Problems and Solutions", Academic Press, 1979.

[Hsia79b] Hsiao, D.K., Kerr, D.S. and Nee, C.J., "Database Access Control in the Presence of Context Dependent Protection Requirements," IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979.

[Hsia80] Hsiao, D.K., "Design Issues of High-Level Language Database Computers," Proceedings of the International Workshop on High-Level Language Computer Architecture, May 26-28, 1980, pp. 92-98, Fort Lauderdale.

[Hsia81] Hsiao, D.K. and Ozsu, T.M., "A Survey of Concurrency Control Mechanisms for Centralized and Distributed Databases," Technical Report, OSU-CISRC-TR-81-1, The Ohio State University, Columbus, Ohio, February 1981.

[Idmsyy] IDMS, Concepts and Facilities, Cullinane Corporation, Wellesley, Mass.

[Jenn77] Jenny, C.J., "Process Partitioning in Distributed Systems," Digest of Papers NTC 1977, 1977.

[Jord81] Jordan, J.J., Banerjee, J. and Batman, R., "Precision Locks," Proceedings of the ACM SIGMOD Conference on Management of Data, April 29 - May 1, 1981, pp. 143-147.

[Kann77a] Kannan, K., "The Design and Performance of a Database Computer," Ph.D. Dissertation, Ohio State University, 1977.

[Kann77b] Kannan, K., Hsiao, D.K. and Kerr, D.S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, October 1977, Niagara Falls, New York, pp. 71-79.

[Kann78] Kannan, K., "The Design of a Mass Memory for a Database Computer," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, Palo Alto, California, pp. 44-50.

[Katz80] Katz, R.H., "Database Design and Translation for Multiple Data Models," Ph.D. Dissertation, University of California, Berkeley, 1980.

[Klei75] Kleinrock, L., "Queueing Systems," Vol. I and II, John Wiley, 1975.

[Knut75] Knuth, D., "The Art of Computer Programming," Vol. III, Addison-Wesley, Reading, Mass., 1975, pp. 475-476.

[Litw78] Litwin, W., "Virtual Hashing : A Dynamically Changing Hashing," Fourth International Conference on VLDB, Berlin, September 1978.

[Liu 75] Liu, M.T. and Reames, C.C., "A Loop Network for the Simultaneous Transmission of Variable Length Messages," Proceedings of the Second Annual Conference on Computer Architecture, January 1975, pp. 7-12.

[Lowe76] Lowenthal, E.I., "The Backend Computer, Part I and Part II," Auerbach (Data Management) Series, 24-01-04 and 24-01-05, 1976.

[Mary76] Maryanski, F.J., Fisher, P.S. and Wallentine, V.E., "A User-Transparent Mechanism for the Distribution of a CODASYL Data Base Management System," Technical Report, TR CS 76-22, Kansas State University, December 1976.

[Mary77] Maryanski, F.J., "Performance of Multi-processor Backend Database Systems," Conference on Information Sciences and Systems, August 1977, pp. 437-441.

[Mary80] Maryanski, F.J., "Backend Database Systems," Computing Surveys, Vol. 12, No. 1, March 1980, pp. 3-25.

[Metc76] Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM, Vol. 19, pp. 395-404, July 1976.

[McCa75] McCauley, E.J., "A Model for Data Secure Systems," Ph.D. Dissertation, Ohio State University, Columbus, Ohio, 1975.

[Meno80] Menon, M.J. and Hsiao, D.K., "The Impact of Auxilliary Information and Update Operations on Database Computer Architecture," International Congress on Applied Systems Research and Cybernetics, December 12-16, 1980.

[Miss80] Missikoff, M. and Terranova, M., "An Overview of the Project DBMAC for a Relational Database Machine," Unpublished technical report, IASI-CNR, personal communication.

[Moha81] Mohan, C., Fussell, D. and Silberschatz, A., "Concurrency, Compatibility, and Commutativity in non-two-phase Locking Protocols," Unpublished, Department of Computer Sciences, University of Texas, Austin.

[Reis79] Reiser, M., "Mean Value Analysis of Queueing Networks, A New Look at an Old Problem," Performance of Computer Systems, North-Holland, 1979.

[Rive76] Rivest, R.L., "Partial-Match Retrieval Algorithms," SIAM Journal of Computing, Vol. 5, No. 1, March 1976, pp. 19-50.

[Rood79] Roode, J.D., "Multiclass Operational Analysis of Queueing Networks," Performance of Computer Systems, North-Holland, 1979.

[Rose77a] Rosenthal, R.S., "An Evaluation of Data Base Management Machines," Annual Computer Related Information System Symposium, U.S. Air Force Academy, 1977.

[Rose77b] Rosenthal, R.S., "The Data Management Machine, a Classification," Workshop on Computer Architecture for Non-Numeric Processing, May 1977, pp. 35-39.

[Roth74] Rothnie, J.R. and Lozano, T., "Attribute Based File Organization in a Paged Memory Environment," CACM, Vol. 17, No. 2, February 1974, pp. 63-69.

[Roth80] Rothnie, J.R., et al, "Introduction to a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, Vol. 5, No. 1, March 1980, pp. 1-17.

[Saat61] Saaty, L., "Elements of Queueing Theory with Applications," McGraw-Hill, New York, 1961.

[Sama80] Samari, N.K. and Schneider, M.G., "A Queueing Theory Based Analytical Model of a Distributed Computer Network," IEEE Transactions on Computers, Vol. C-29, No. 11, Nov. 1980.

[Sava76] Savage, J.E., "The Complexity of Computing," John Wiley, New York, 1976.

[Schn73] Schneiderman, B., "Optimum Database Reorganization Points," CACM, Vol. 16, No. 6, June 1973, pp. 362-365.

[Schu79] Schuster, S.A., Nguyen, H.G., Ozkarahan, E.A. and Smith, K.C., "RAP.2 - An Associative Processor for Databases and its Applications," IEEE Transactions on Computers, Vol. C-28, No. 6, June 1979, pp. 446-458.

[Song80] Song, S.W., "A Highly Concurrent Tree Machine for Database Application," International Conference on Parallel Processing, 1980, pp. 259-268.

[Stea76] Stearns, R.E., Lewis, P.M. II and Rosenkrantz, D.J., "Concurrency Control for Database Systems," Proceedings 17th IEEE Symposium on Foundations of Computer Science, October 1976, pp. 19-32.

[Ston74] Stonebraker, M.R. and Wong, E., "Access Control in a Relational Data Base Management System by Query Modification," Proceedings of ACM Annual Conference, San Diego, California, November 1974, pp. 180-187.

[Ston76a] Stonebraker, M. and Neuhold, E., "A Distributed Data Base Version of INGRES," Proceedings of the Second Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, California, May 1976.

[Ston76b] Stonebraker, M. and Rubinstein, P., "The INGRES Protection System," ACM Annual Conference, 1976, pp. 80-84

[Ston78] Stonebraker, M., "A Distributed Data Base Machine," Memorandum No. UCB/ERL M78/23, Electronics Research Laboratory, University of California, Berkeley, May 1978.

[Ston79] Stonebraker, M., "Muffin: A Distributed Data Base Machine," First International Conference on Distributed Computing Systems, 1979, pp. 459-469.

[Tane81] Tanenbaum, A.S., "Computer Networks," Prentice-Hall Inc., 1981.

[Systyy] SYSTEM 2000 Reference Manual, MRI Systems Corporation, Austin, Texas.

[Thom79] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.

[Totayy] TOTAL Reference Manual, Cincom Systems Corporation, Cincinnati, Ohio.

[Wah 80] Wah, B.W. and Yao, B.S., "DIALOG - A Distributed Processor Organization for Database Machine," Proceedings of the National Computer Conference, 1980, pp. 243-253.

[Wong71] Wong, E. and Chiang, T.C., "Canonical Structure in Attribute Based File Organization," CACM, Vol. 14, No. 9, September 1971, pp. 593-597.

APPENDIX G: DETERMINING CLUSTERS CORRESPONDING TO AN ARBITRARY QUERY

In this appendix, we will describe the algorithm for determining the set of clusters corresponding to an arbitrary query. This algorithm employs the other algorithm described in Chapter 4 for determining the set of clusters corresponding to a query in disjunctive normal form. For simplicity, we will refer to the algorithm for determining the set of clusters corresponding to an arbitrary query as Algorithm Query. Similarly, we will refer to the algorithm for determining the set of clusters corresponding to a query in disjunctive normal form as Algorithm Disjunct. Algorithm Query takes an arbitrary query and generates the corresponding set of clusters for the query as output. Algorithm Disjunct takes a query in disjunctive normal form as input and generates the corresponding set of clusters for the query as output. Thus, both algorithms will have two arguments. The first argument is the input query. The second argument is the output set of clusters.

```
Algorithm Query
/*Q is the input query*/
/*D is the set of clusters output for Q*/
Read next character X from Q;
Q' = unread portion of Q;
If X = '(' then
    begin
        Call Query (Q',D1);
        Read next character of Q;
    end
    else
    begin
        Read the next predicate P of Q;
        Call Disjunct(P,D1);
    end
again:
Read next character X from Q;
Q' = remaining portion of Q until first unread 'V' or '∧';
If X = 'V' then
    begin
        Call Query(Q',D2);
        D=D1 ∪ D2;
    end
else if X = '∧' then
    begin
        Call Query(Q',D2);
        D=D1 ∩ D2;
    end
else stop;
Go to again;
End Query
```

APPENDIX H: A DESCRIPTION OF THE SIMULATION MODEL

Our simulation model of MDBS follows the so-called scenario approach [Fran77] to simulation. Thus, MDBS is simulated as a number of scenarios or processes. Consider an MDBS system with n back-ends and m disk drives. Then, MDBS is simulated by (n + m + 3) processes. These are the processes for the n back-ends, the m disk drives, the controller, the broadcast bus and the VAX Unibus. Each process in the simulation model is associated with a queue. The queue associated with a back-end is referred to as a back-end queue; the queue associated with a disk drive is referred to as a disk drive queue; the queue associated with the controller is referred to as the controller queue; the queue associated with the broadcast bus is referred to as the broadcast bus queue and the queue associated with the VAX Unibus is referred to as the VAX Unibus queue.

Each of the above queues consists of zero or more elements. An element in one of these queues may be of one of several types. For instance, the controller queue contains three types of elements. First, some elements are requests which have to be parsed, processed, and so on. In the simulation model, these elements are termed of type 1. Second, in the controller queue some elements are responses received from the back-ends. In our model, these elements are of type 6. Finally, the queue contains type-4 elements. These are messages sent from the back-ends to the controller to indicate the cluster number of a record for insertion.

A back-end queue contains three types of elements. First, it contains requests on which directory processing has to be performed by the back-end. These are type-2 elements. After a back-end performs directory processing on a request, it broadcasts the corresponding descriptors to all the remaining back-ends. Thus, it contains descriptors which have been broadcasted from other back-ends after directory processing. The corresponding descriptors are stored as type-5 elements in the back-end queues. Third, a back-end queue will contain elements of type 3 which are messages broacasted by the controller after consultation of the cluster-id-to-next-back-end-table (CINBT) for an insert request.

A disk drive queue consists only of a single type of element. These elements are I/O requests generated by the back-end to which the disk drive is attached.

The broadcast bus queue consists of elements of five types. First, it consists of requests which are initially broadcasted from the controller to the back-ends via the broadcast bus. These are elements of type 2. Second, it contains type-3 elements consisting of the messages from the controller to the back-ends after consultation of the CINBT for an insert request. Third, it contains type-5 elements made of the corresponding descriptors for a request which are broadcasted from a back-end to all other (n-1) back-ends. Fourth, it contains type-6 elements, i.e., the responses sent from the back-ends to the controller. Finally, it contains type-4 elements consisting of the messages from the back-ends to the controller indicating the cluster for insertion.

The VAX Unibus queue contains elements of six different types. It contains elements of types 2, 3, 4 and 6 as described above for the broadcast bus queue. In addition, it consists of type-1 elements made of requests initially submitted to MDBS. It also consists of type-7 elements, i.e., the outputs of MDBS.

A DESCRIPTION OF PROCESSES

We are now ready to describe the (n + m + 3) processes which cumulatively characterize the simulation model of MDBS. Since the n processes which model the n different back-ends are identical, and since the m processes which model the m different disk drives are identical, we only need to describe five processes. These five processes will be described below. These processes will utilize the various queues described in the previous section.

Basically, the following is done in each process. The next element from the queue associated with the process is examined. The kind of service to be performed on that element will depend on the type of the element. Having determined the type of the element, the appropriate service is performed for that element by advancing the simulation clock for the time needed to perform (i.e., <u>delaying for time to perform</u>, in simulation terminology) that service. The service may also include placing the element in another queue. After performing the appropriate service, the element is removed from this queue and the next element in this queue is examined. The above procedure is repeated until the simulation termination criterion is met. In our model, the termination criterion is met when the simulation clock has advanced to a very large value. This completes the basic description of each process. What varies from process to process is the kind of service which needs to be performed on the elements in the appropriate queues associated with these processes. Let us now describe the kinds of services which need to be performed for each of the processes, in turn.

A. The Back-end Process

Step 1: Pick the next element from the back-end queue. If the queue is empty, then go to step 11.

Step 2: Examine the type of the element. If element is of type 2, then go to step 3 and perform directory processing. If element is of type 5, then go to step 4 and accept corresponding descriptors. If element is of type 3, then go to step 10 and perform record insertion.

Step 3: Delay for time to perform descriptor processing. Change the type of the element to 5 and insert it into the broadcast bus queue. Go to step 1.

Step 4: Accept corresponding descriptors. Check if the corresponding descriptors for this request have been received from all the other (n-1) back-ends. If the request is an insert request, then delay for time to perform the first step of address generation. Generate an element of type 4 to represent a message from the back-end to the controller which indicates the cluster number of a record for insertion and go to step 1. If the request is a non-insert request, then delay for time to perform the three steps of address generation. Access the tracks containing the records of the permitted set of clusters by placing the i/o requests into disk drive queues. Wait until a back-end indicates that a track of records has been accessed and placed in the main memory of the back-end. Then, process each such accessed record as in steps 5 through 9.

Step 5: Check each accessed record against the user query. Delay for time to check a record against a single predicate multiplied by the number of predicates in the user query.

Step 6: If the request is a retrieve request, then go to step 7. If the request is an update request, go to Step 8. If the request is a delete request, go to step 9.

Step 7: If the record satisfies the user query, place it in a main memory buffer for output to the controller. Go to step 5 if there are more records to be retrieved. Else, return the records in the buffer to the controller and go to step 1.

Step 8: If the record satisfies the user query, update the record and place it in a main memory buffer for writing back to a disk drive. If the main memory buffer is full, initiate an i/o request to write the contents of the buffer to the secondary store. Go to step 5 if there are more records to be updated. Else, send an update-completion message to the controller and go to step 1.

Step 9: If the record satisfies the user query, mark the record with deletion tag and place it in a main memory buffer for writing back to a disk drive. If the main memory buffer is full, initiate an i/o request to write the contents of the buffer to the secondary store. Go to step 5 if there are more records to be deleted. Else, send a delete-completion message to the controller and go to step 1.

Step 10: Delay for the time to perform the second and third steps of address generation. Then, generate an i/o request to the disk drive selected for inserting the record. On receipt of message from disk drive indicating successful completion of insertion, send an insert-completion message to the controller and go to step 1.

Step 11: Wait for next element to arrive to the back-end queue. When element arrives, go to step 1.

B. The Disk Drive Process

Step 1: Pick the next element from the controller queue. If queue is empty, go to step 5.

Step 2: Examine the track number and cylinder number to which the head must be moved to complete the i/o request represented by the element. Delay for time to move the head to the appropriate track.

Step 3: Wait until back-end unibus becomes free.

Step 4: Delay for track-rotation time. Go to step 1.

Step 5: Wait for next element to arrive to the disk drive queue. When element arrives, go to step 1.

C. The Controller Process

Step 1: Pick the next element from the controller queue. If queue is empty, go to step 6.

Step 2: Examine the type of the element. If the element is of type 1, then go to step 3. If the element is of type 4, then go to step 4. If the element is of type 6, then go to step 5.

Step 3: Delay for time to parse the request and to broadcast it to all the back-ends. Place a type-2 element in the VAX Unibus queue for eventual transmission to the n back-ends. Go to step 1.

Step 4: Check if (n-1) such elements have already been processed. If not, then go to step 1. Else, delay for time to search CINBT. Generate a type-3 element to represent a message to be sent to the back-end chosen for record insertion. Place this element in the VAX Unibus queue. Go to step 1.

Step 5: Check if (n-1) such elements have already been received. If not, then go to step 1. Else, generate a type-7 element to represent the response to be returned to the user and place it in the VAX Unibus queue. Go to step 1.

Step 6: Wait for next element to arrive to the controller queue. When it arrives, go to step 1.

D. The Broadcast Bus Process

Step 1: Pick the next element from the broadcast bus queue. If queue is empty, go to step 7.
Step 2: If the type of the element is 4 or 6, go to step 3. If the type of the element is 5, go to step 4. If the type of the element is 3, go to step 5. Else, go to step 6.
Step 3: Place the element in the VAX Unibus queue. Delay for bus transmission time. Go to step 1.
Step 4: Generate (n-1) elements of type 5 to be stored in the queues of the (n-1) back-ends other than the one which placed the element in the broadcast bus queue. Delay for bus transmission time. Go to step 1.
Step 5: Place the element in the queue associated with the back-end chosen for insertion. Delay for bus transmission time. Go to step 1.
Step 6: Generate n elements of type 2 for the queues of all the n back-ends. Delay for bus transmission time. Go to step 1.
Step 7: Wait for the next element to arrive to the broadcast bus queue. When it arrives, go to step 1.

E. The VAX Unibus Process

Step 1: Pick the next element from the VAX Unibus queue. If queue is empty, go to step 6.
Step 2: If the type of the element is 4, 6 or 1, then go to step 3. If the type of the element is 7, then go to step 4. Else, go to step 5.
Step 3: Delay for VAX Unibus transmission time. Place the element into the controller queue. Go to step 1.
Step 4: Delay for VAX Unibus transmission time. Go to step 1.
Step 5: Delay for VAX Unibus transmission time. Place the element into the broadcast bus queue. Go to step 1.
Step 6: Wait for next element to arrive to the VAX Unibus queue. When it arrives, go to step 1.

APPENDIX I: INPUT PARAMETERS OF THE SIMULATION MODEL

The following are the input parameters of the simulation model used in Chapter 7. Each of these parameters has to be specified for every simulation run.

A. Parameters Related to Requests

Percentage of retrieve requests
Percentage of update requests
Percentage of delete requests
Percentage of insert requests
Interarrival time of requests
Number of predicates in the query of a request
Number of clusters to be processed for a request

B. Parameters Related to the Disk Drives

Seek time
Rotation time
Number of disk drives per back-end
Number of cylinders per disk drive
Number of tracks per cylinder
Number of bytes per track

C. Parameters Related to the Databases

Number of clusters
Average cluster size

D. Parameters Related to the System

Number of back-ends
Broadcast bus speed
VAX Unibus speed
Time to send a message
Time to parse a request
Time to check if a record satisfies a predicate